

10-30-00

"Express Mail" Mailing Label No. EL 521 771 105 US

Date of Deposit October 27, 2000

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above, with sufficient postage affixed, and is addressed to BOX PATENT APPLICATION, Assistant Commissioner for Patents, Washington, D.C. 20231.

Lucy Flemings

(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

Patent

Attorney's Docket No. 032001-074

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

UTILITY PATENT
APPLICATION TRANSMITTAL LETTER

Box PATENT APPLICATION

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

Enclosed for filing is the utility patent application of Daniel J. Pugh and Mark Rollins for GOLD CODE GENERATOR DESIGN.

Also enclosed are:

- ☒ 9 sheet(s) of ☐ formal ☒ informal drawing(s);
- ☐ a claim for foreign priority under 35 U.S.C. §§ 119 and/or 365 is ☐ hereby made to filed in on ;
- ☐ in the declaration;
- ☐ a certified copy of the priority document;
- ☐ a General Authorization for Petitions for Extensions of Time and Payment of Fees;
- ☒ an Assignment document;
- ☒ an Information Disclosure Statement; and
- ☒ Other: Appendix

☒ A ☐ executed ☒ partially executed declaration of the inventor(s)

☒ also is enclosed ☐ will follow.

☐ Please amend the specification by inserting before the first line the sentence --This application claims priority under 35 U.S.C. §§ 119 and/or 365 to filed in on ; the entire content of which is hereby incorporated by reference.--

☒ Small entity status is hereby claimed.



21839

(10/00)

[X] The filing fee has been calculated as follows [] and in accordance with the enclosed preliminary amendment:

C L A I M S					
	No. Of CLAIMS		EXTRA CLAIMS	RATE	FEE
Basic Application Fee					\$710.00 (101)
Total Claims	22	MINUS 20 =	2	× \$18.00 (103) =	36.00
Independent Claims	4	MINUS 3 =	1	× \$80.00 (102) =	80.00
If multiple dependent claims are presented, add \$270.00 (104)					
Total Application Fee					826.00
If small entity status is claimed, subtract 50% of Total Application Fee					413.00
Add Assignment Recording Fee \$ if Assignment document is enclosed					40.00
TOTAL APPLICATION FEE DUE					453.00

[] This application is being filed without a filing fee. Issuance of a Notice to File Missing Parts of Application is respectfully requested.

[X] A check in the amount of \$ 453.00 is enclosed for the fee due.

[] Charge \$ _____ to Deposit Account No. 02-4800 for the fee due.

[X] The Commissioner is hereby authorized to charge any appropriate fees under 37 C.F.R. §§ 1.16, 1.17 and 1.21 that may be required by this paper, and to credit any overpayment, to Deposit Account No. 02-4800. This paper is submitted in duplicate.

Please address all correspondence concerning the present application to:

Robert E. Krebs
BURNS, DOANE, SWECKER & MATHIS, L.L.P.
P.O. Box 1404
Alexandria, Virginia 22313-1404.

Respectfully submitted,

BURNS, DOANE, SWECKER & MATHIS, L.L.P.

Date: October 27, 2000

By:

Joseph P. O'Malley
Registration No. 36,226

P.O. Box 1404
Alexandria, Virginia 22313-1404
(650)622-2300

"Express Mail" Mailing Label No. EL 521 771 105 US

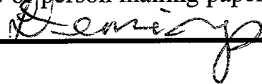
Date of Deposit October 27, 2000

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above, with sufficient postage affixed, and is addressed to BOX PATENT APPLICATION, Assistant Commissioner for Patents, Washington, D.C. 20231.

Lucy Flemings

(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)



PATENT

Atty Dkt. No. 032001-074

GOLD CODE GENERATOR DESIGN

BACKGROUND OF THE PRESENT INVENTION

The present application concerns pseudo-random generators, in particular gold code generators.

- 5 Pseudo-random generators have applicability for a number of communication systems, in particular, for spread spectrum wireless communications. In spread spectrum transmissions, the circuit artificially spreads the transmitted signals bandwidth by modulating an information signal, either in phase or frequency, with a pseudo-random sequence that occurs at a greater rate
- 10 than that required for the data alone. During signal reception, the receiver synchronizes an internal pseudo-random generator to the pseudo-random sequence of the transmitted signal to fully recover the available power and decode the message. Most direct sequence spread spectrum systems pseudo-randomly modulate the phase of the RF carrier signal 10 times or greater than the rate
- 15 required for the data transmission. This results in a signal spectrum which is much broader than would be occupied if the RF carrier signal were modulated by only the data stream. Frequency hopping systems use the pseudo-random generator to implement frequency hops within the spread spectrum range.

Matching pseudo-random generators at the transmitter and receiver allow the correlation and recovery of the information signal. Other transmitted signals with different pseudo-random codes can be transmitted in the same bandwidth since the correlation between the different pseudo-random codes is quite low. The transmissions using the different pseudo-random codes will tend not to significantly interfere with one another.

One way of implementing a pseudo-random generator is with a linear feedback shift register (LFSR). Taps from the linear feedback register are sent to a logic circuit to create a new input (feedback) bit. The linear shift register runs through a large number of different codes before repeating. The linear feedback shift register is preferably selected with a feedback path producing the maximum code length. Also beneficial for the linear shift register is low auto-correlation with shifts in the pseudo-random sequence and low cross-correlation with other sequences.

One preferred way of implementing a pseudo-random sequence is to combine the outputs of two linear feedback shift registers. Such a pseudo-random generator is called a gold code generator.

Figure 1 illustrates a gold code generator used with the UMTS European wireless standard. The gold code generator is constructed of two linear feedback shift registers. The first linear feedback shift register has feedback taps from registers 0 and 3. The second linear feedback shift register has feedback taps at registers 0, 1, 2, and 3. The first serial linear feedback shift register's output is combined with the output of the second linear feedback shift register in the EXCLUSIVE-OR 40. The first linear feedback shift register 42 has taps at registers 4, 7 and 18 that go to a EXCLUSIVE-OR (mask) 44. The second linear feedback shift register 46 has taps at registers 4, 6, and 17 fed to the EXCLUSIVE-OR 48. The output to the EXCLUSIVE-ORs 44 and 48 are sent to a second output EXCLUSIVE-OR 52.

It is desired to have an improved implementation of a gold code generator.

SUMMARY OF THE PRESENT INVENTION

The inventors have noticed that the range of taps used to implement the second output of the UMTS gold code generator standard is quite broad: In the first linear shift register between taps 4 and 18 and in the second linear shift register between taps 4 and 17. This broad range of taps makes it difficult to implement a parallel implementation of the gold code generator in arithmetic logic units or other computational units that operate on parallel data.

Since the second output is in fact a delayed version of the first output, the gold code generator can be implemented by forming two pairs of linear feedback shift registers and using different seeds for the second pair of linear feedback shift registers. This significantly reduces the range of the output taps in any of the linear feedback shift registers, and makes it easier to implement a parallel implementation of the gold code generator to produce multiple output bits.

One embodiment of the present invention comprises a gold code generator comprising two pairs of linear feedback shift registers wherein the second seed values for the second pair of linear feedback shift registers are different from the first seed values for the first pair of linear feedback state machines. The second seed values are calculated from the first seed values, wherein the first and second pair of linear feedback shift registers are implemented to produce more than one input bit and more than one output bit for each linear feedback shift registers at the same time.

Another embodiment of the present invention comprises at least one reconfigurable chip implementing a gold code generator, the at least one reconfigurable chip including background and foreground configuration memories. The background configuration memory is adapted such that it can be loaded with a gold code generator configuration while the at least one reconfigurable chip

configured with the foreground configuration operates. After the background configuration is loaded with the gold code generator configuration, the background plane is activated to reconfigure the at least one reconfigurable chip.

Another embodiment of the present invention comprises a method of
5 implementing a pseudo-random code generator comprising the steps of converting a pseudo-random code generator specification into an equivalent representation. The pseudo-random code generator specification being such that taps used to calculate an output include at least one tap within n spaces of the input. Equivalent representation is such that no taps are within n spaces from the input. The method
10 includes implementing the equivalent representation such that n new state bits are calculated at the same time.

Another embodiment of the present invention is a method of implementing a pseudo-random code generator, the method comprising converting a pseudo-random code generation specification into an equivalent of representation. The
15 pseudo-random code generator specification being such that taps used to calculate an output bit are defined within a first shift register span, the equivalent representation is such that taps used to calculate an output bit within a smaller shift register span. The method includes implementing the equivalent representation such that multiple new bit states are calculated at the same time.

20 BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a diagram of a prior art gold code generator.

Figure 2 shows the diagram illustrating the implementation of the gold code generator of Figure 1 into an equivalent representation.

Figure 3 is a diagram of that illustrates a parallel implementation of the
25 gold code generator.

Figure 4 is a diagram of a functional block diagram of a gold code generator.

Figure 5 is a diagram of a reconfigurable chip which can be used for implementing the gold code generator of the present invention.

Figure 6A and Figure 6B show a method of switching in the gold code configuration used with one embodiment of the present invention into a reconfigurable fabric of a reconfigurable chip.

Figure 7 is a diagram illustrating the Galois field calculations for the seed values.

Figure 8A and Figure 8B are tables illustrating the values of the lookup tables of Figure 3.

10 DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 2 illustrates the conversion of the gold code generator 60 as defined in the UMTS specification into an equivalent representation using two pairs of linear feedback shift registers, pairs 62 and 64. The output from unit 66 of the gold code generator 60 is equivalent to a delayed sequence of the output unit 68.

15 In the present invention, multiple pairs of the linear feedback shift register are used, the second pair of linear feedback shift registers uses a second pair of seed values such that output of the second pair of linear feedback shift registers 64 is a delayed sequence of the sequence produced by the output 66 of the gold code generator 60.

20 Although the equivalent representation uses more resources, this equivalent representation can be implemented in a parallel implementation that produces multiple output bits. Such a representation is especially useful when implemented with a reconfigurable chip in which reconfigurable elements are configured by configuration bits. One reason why it is easier to do a parallel implementation of

25 the equivalent representation is that a narrow range of output taps is used with the equivalent representation. The output 66 of the gold code generator 60 is quite broad. This makes it difficult to calculate multiple output bits with the standard

gold code representation. In the equivalent representation, the linear feedback shift register pair 64 has only two output bits going to the EXCLUSIVE-OR 70. Additionally, the taps used to produce the feedback states remain relatively close together for both of the pairs of the linear feedback shift registers.

5 The second shift register 64 is seeded with a new initial seed. This new initial seed value can be calculated before operation of the gold code generator. This calculation is a Galois field calculation which can be done as shown in Figure 7. Additional discussion of the Galois field calculation is given in Chapter 6 entitled "Theory and application of pseudo-random sequences" in the reference
10 "CDMA Systems Engineering Handbook" by Lee and Miller, which is incorporated herein by reference.

 Additionally, a copy of the C code to calculate the seed values including some corrections to the math of the Lee and Miller book are enclosed as Appendix II to this application.

15 In a preferred embodiment, each of the linear feedback shift registers is reset to an initial value at the beginning of each frame. The seed for the linear feedback shift register LFSRA is assigned by the network controller for each user. The seed for the linear feedback shift register LFSRB is 0 x 1FFFFFF for all users. The seed for the linear feedback shift register LFSRC is the contents of the linear
20 feedback shift register LFSRA seed shifted by 16,777,232,000 cycles, and can be computed by a processor at the beginning of each call. The seed for linear feedback shift register LFSRD is 0 x 1FFFFFF shifted by 16,777,232,000 cycles for each user and thus is a static constant value which can be precalculated and stored. The seed for all of the linear feedback shift registers can be stored in
25 memory elements in a reconfigurable fabric.

 Figure 3 shows a parallel implementation of one of the linear feedback shift registers of the equivalent representation. The span of taps to produce an output bit is quite small so no complicated logic is required to calculate an output. The

input bits are also calculated in parallel using lookup tables. Note how a parallel implementation spreads the number of taps needed for the feedback or output calculations. There is no requirement for a really wide span lookup table for the output bits since the equivalent representation is used.

5 Figures 8A and 8B show the table lookup values of one embodiment of the lookup tables of Figure 3.

Figure 4 illustrates a functional block diagram of a gold code generator in one embodiment. The gold code generator of one embodiment deals with multiple users, each user having a different input seed thus producing different sequence of
10 the gold code generator, each of the different sequences having a relatively low cross-correlation. In this embodiment, for each seed the delayed version for the second pair of shift registers must be calculated using the Galois field calculation described above.

Figure 5 illustrates a reconfigurable chip 80. The reconfigurable chip 80
15 implements the gold code generator in one embodiment. The reconfigurable chip 80 includes a reconfigurable fabric 82 which can be configured into a variety of configurations. The CPU 88 can be used for the LFSRC seed calculations that are difficult to do in the reconfigurable fabric.

In a preferred embodiment, the reconfigurable chip 80 includes the
20 processor 88 such as a reduced instructions set computing (RISC) central processing unit (CPU). In one embodiment the CPU 88 runs portions of the algorithms which are difficult to implement in the reconfigurable fabric.

In one embodiment, the reconfigurable fabric is configured by a foreground configuration plane 84. While the foreground configuration plane 84 is operating,
25 the background configuration can be loaded from the background plane 86. The reconfigurable fabric 82 in a preferred embodiment includes a number of configurable data path units, memory elements, and interconnect elements.

In one embodiment, the data path units include comparators, an arithmetic unit (ALU), and registers which are configurable to implement operations of the algorithm. In one embodiment the reconfigurable fabric 82 also includes dedicated elements such as multiple elements and memory elements. The memory elements
5 can be used for storing algorithm data.

Figures 6A and 6B illustrate the operation of the reconfigurable fabric to the system of the present invention. In Figure 6A, the gold code generator configuration is loaded into the background configuration plane 90. The foreground configuration plane 92 is configured with another configuration so as
10 to configure the reconfigurable fabric with that configuration. In Figure 6B the configuration plane in background plane is loaded into the foreground plane. This almost instantaneously configures the reconfigurable fabric 92' into the gold code generator configuration.

Details of the implementation of the gold code generator are shown in
15 Appendix I, especially on pages 42-64.

Although only preferred embodiments of the invention are specifically disclosed and described above, it will be appreciated that many modifications and variations of the present invention are possible in light of the above teachings and within the purview of the appended claims without departing from the spirit and
20 intended scope of the invention.

Claims:

1. A gold code generator comprising:
two pairs of linear feedback shift registers, wherein second seed values for the second pair of linear feedback shift registers are different from first seed values for the first pair of linear feedback shift registers, the second seed values being calculated from the first seed values, wherein the first and second pair of linear feedback shift registers are implemented to produce more than one new state bit and more than one output bit for each linear feedback shift registers at the same time.
2. The gold code generator of claim 1 wherein the seed values for the second pair of linear feedback shift registers are delayed values of the first seed values.
3. The gold code generator of claim 1 wherein the gold code generator is implemented on a reconfigurable logic chip.
4. The gold code generator of claim 3 wherein the calculation of some of the second seed values is done using a dedicated processor on the reconfigurable chip.
5. The gold code generator of claim 3 wherein the gold code generator configuration is loaded into a background plane of the reconfigurable chip, while the reconfigurable chip is operating on another configuration in the foreground.
6. The gold code generator of claim 3 wherein the feedback is implemented using lookup tables.

7. A system comprising:

at least one reconfigurable chip implementing a gold code generator, the at least one reconfigurable chip including background and foreground configuration memories, wherein the background configuration memory is adapted such that it can be loaded with a gold code generator configuration while the at least one reconfigurable chip, configured with the foreground plane, operates, and wherein after the background configuration loads, the gold code generator configuration can be activated to reconfigure the at least one reconfigurable chip.

8. The system of claim 7 wherein the gold code generator comprises two pairs of linear feedback shift registers wherein the seed values for the second pair of linear feedback shift registers is different from the seed values for the first pair of linear feedback shift registers.

9. The gold code generator of claim 7 wherein the second seed values are calculated from the first seed values.

10. The gold code generator of claim 9 wherein the calculation of the second seed values is done at least partially in a processor on the reconfigurable chip.

11. The system of claim 10 in which the gold code generator is implemented to produce more than one output bit at the same time.

12. A method of implementing a pseudo-random code generator:

converting a pseudo-random code generator specification into an equivalent representation, the pseudo-random code generator specification being such that taps used to calculate an output include at least one tap within n spaces from the

input, the equivalent representation is such that no such taps are within n spaces from the input; and

implementing the equivalent representation such that multiple new state bits are calculated at the same time.

13. The method of claim 12 wherein the pseudo-random code generator specification being such that taps to calculate an output is defined within a first chip register span, the equivalent representation is such that taps to calculate an output bit are within a smaller shift register span.

14. The method of claim 12 wherein the equivalent representation includes two pairs of linear feedback shift registers wherein the second seed values for the second pair of linear feedback shift registers is different from a first seed value for the first pair of linear feedback shift registers.

15. The method of claim 12 wherein the pseudo-random code generator comprises a gold code generator.

16. The method of claim 12 wherein the pseudo-random code generator is implemented on a reconfigurable chip.

17. A method of implementing a pseudo-random code generator comprising:

converting a pseudo-random code generator specification into an equivalent representation, the pseudo-random code generator specification being such that taps to calculate an output are defined within a first shift register span, the equivalent representation is such that the taps to calculate an output bit are within a smaller shift register span; and

implementing the equivalent representation such that multiple output bits are calculated at the same time.

18. The method of claim 17 wherein the pseudo-random code generation specification is such that taps used to calculate an output have at least one tap within n spaces from the input, the equivalent representation is such that no such tap is within n spaces from the input.

19. The method of claim 17 wherein two pairs of linear feedback shift registers are used in the equivalent representation.

20. The method of claim 19 wherein the second seed values for the second pair of linear feedback shift registers are different from the first seed values for the first pair of linear feedback shift registers.

21. The method of claim 17 implemented on a reconfigurable chip.

22. The method of claim 17 wherein in the equivalent representation of the output bits are calculated from taps at a single register for each linear feedback shift register.

Abstract

- A gold code generator is described comprising two pairs of linear feedback shift registers, the seed values for the second pair of linear feedback shift registers are different from the seed values for the first pair of linear feedback shift registers. The second seed values are calculated from the first seed values. The use of this second pair of linear feedback shift registers prevents the need to use a wide span of taps to the linear feedback shift register to produce output bits. By using two pairs of linear feedback shift registers, a parallel output implementation can be produced in which multiple output bits are produced in a single clock cycle.
- 5

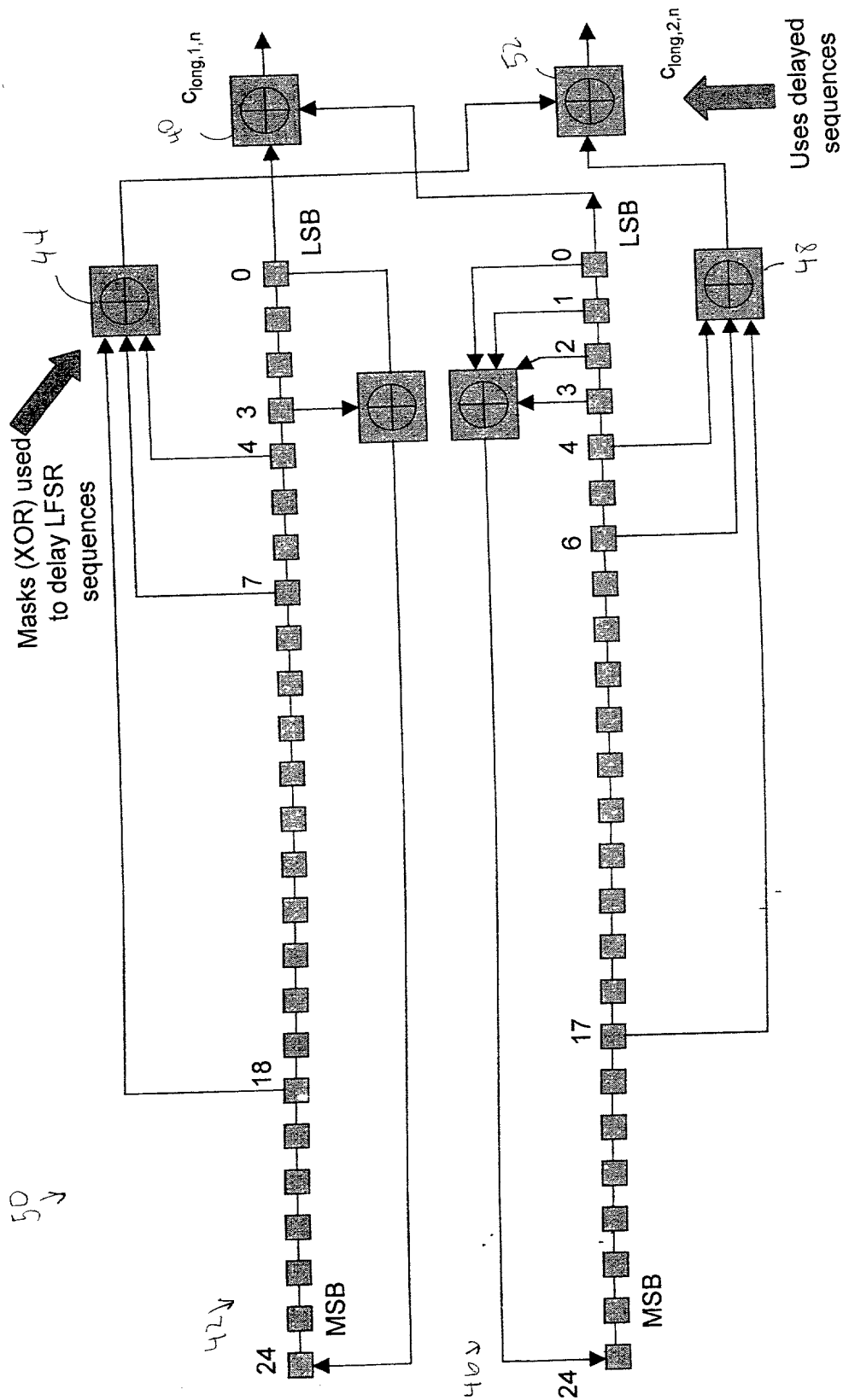


FIGURE 1
(PRIOR ART)

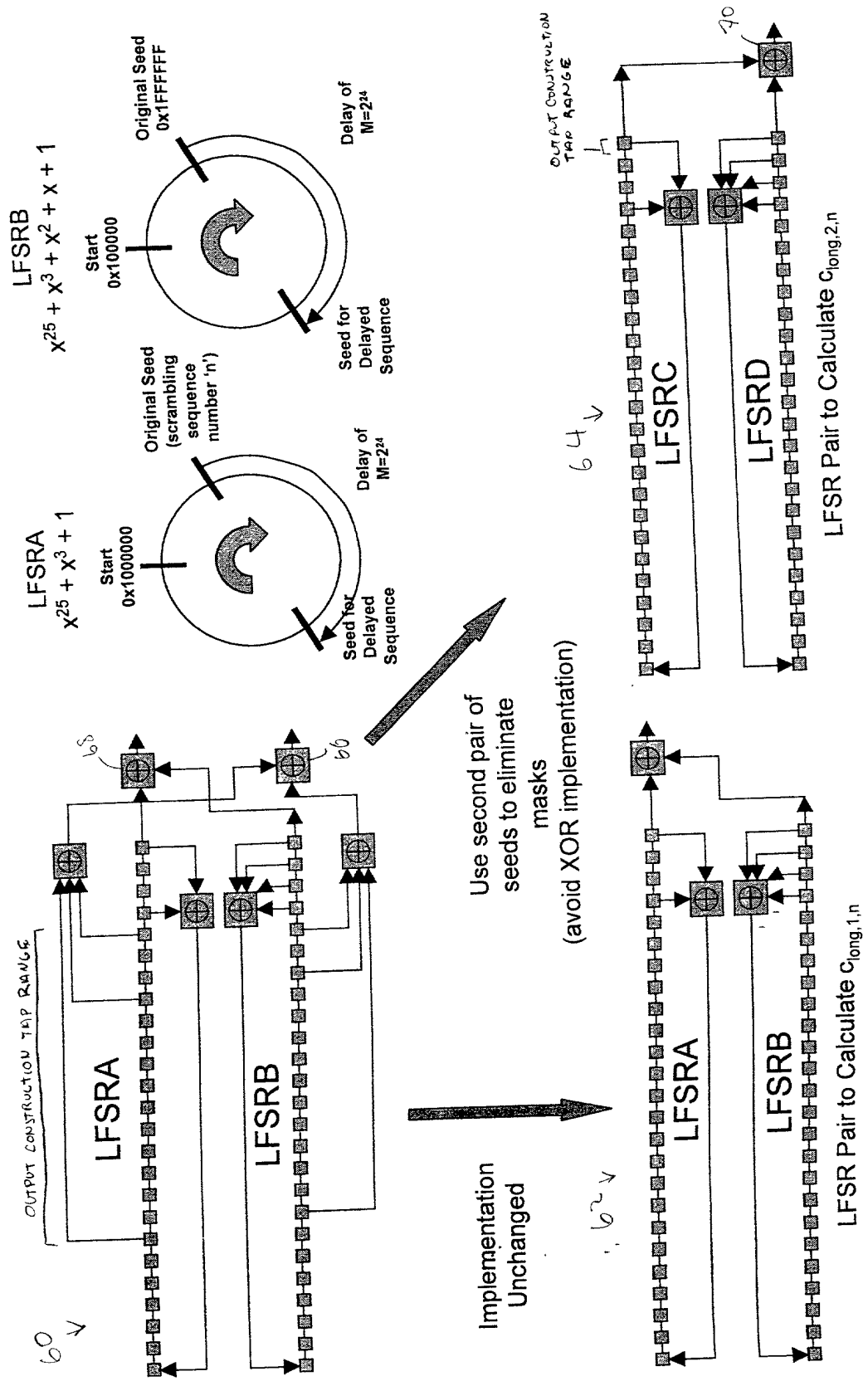


FIGURE 2

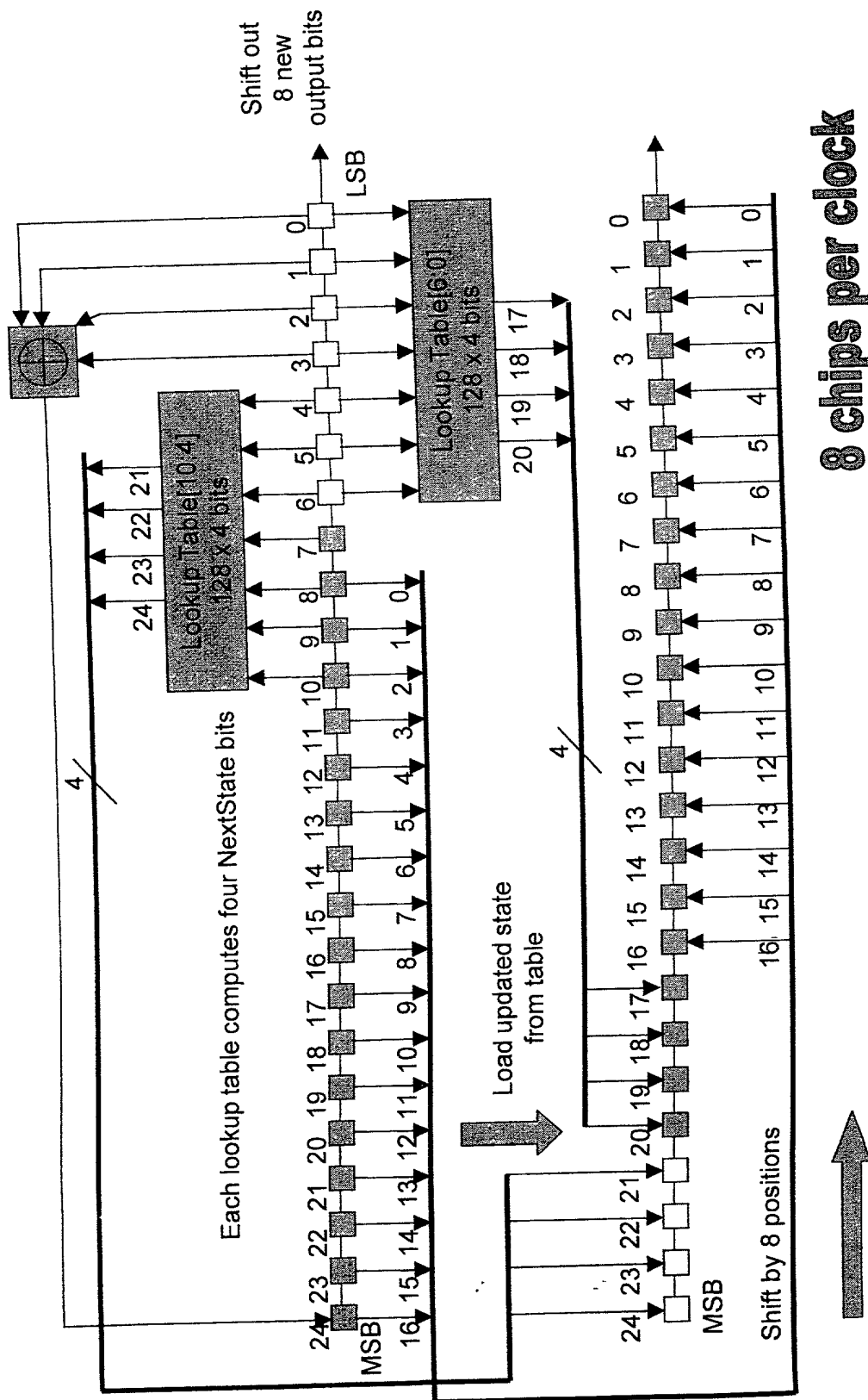


FIGURE 3

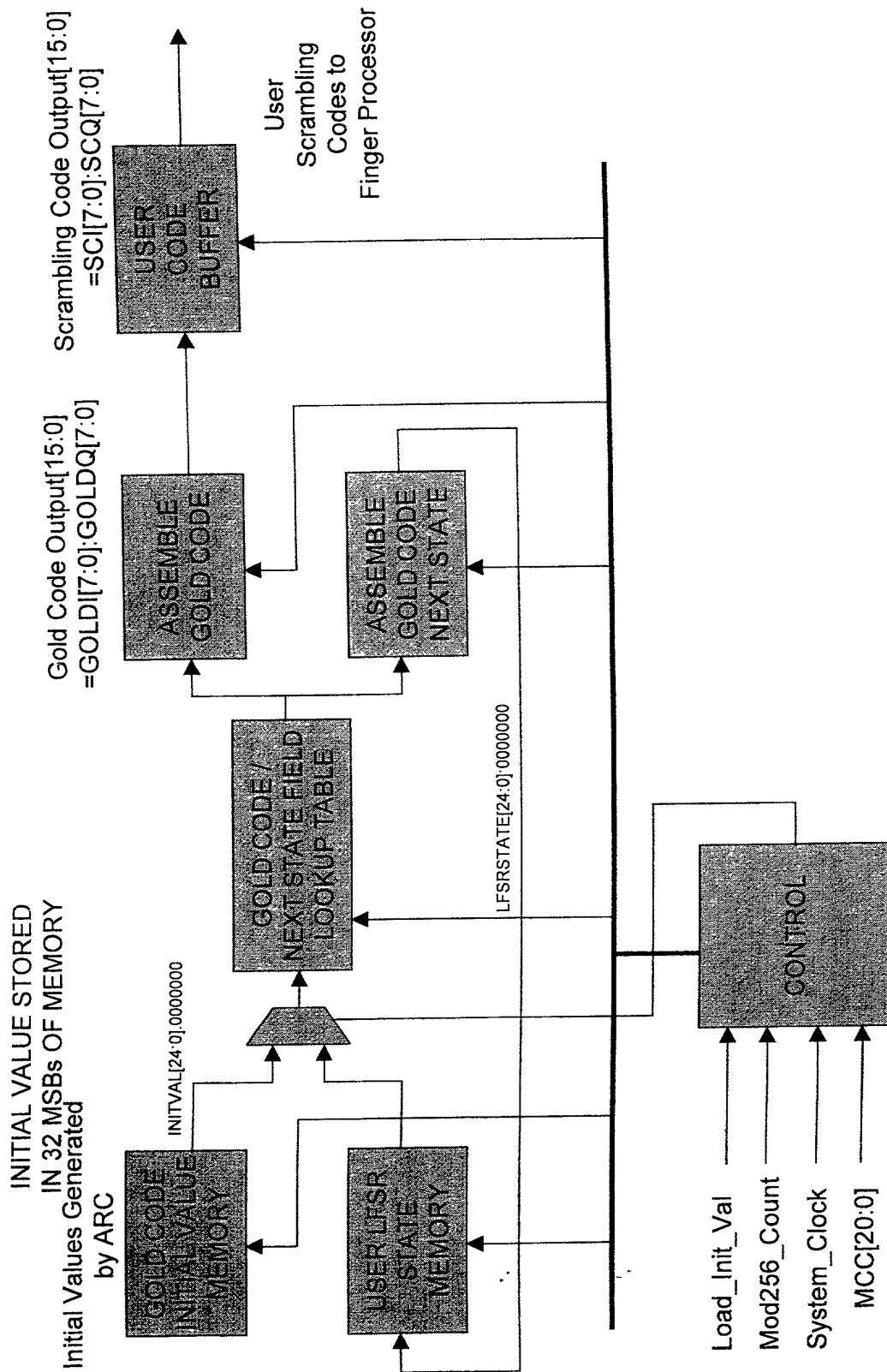


FIGURE 4

2025 RELEASE UNDER E.O. 14176

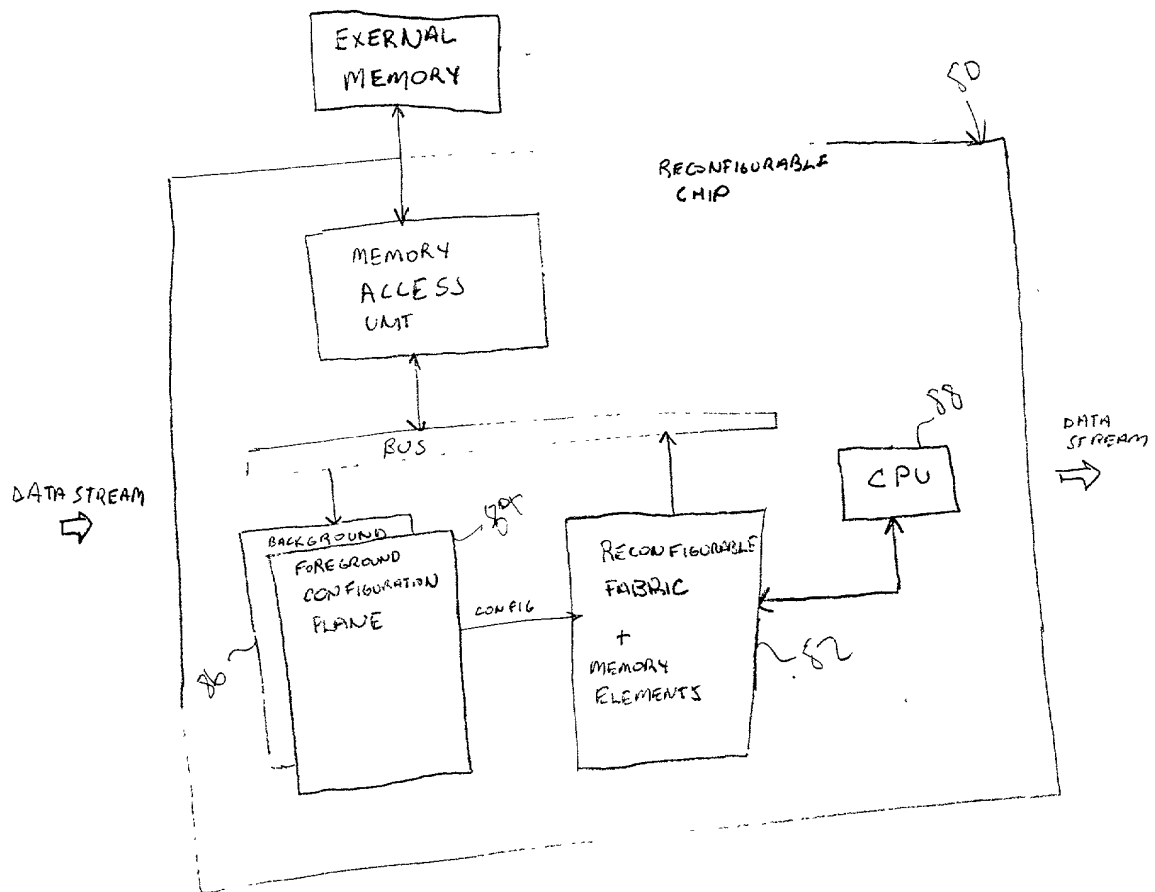


FIGURE 5

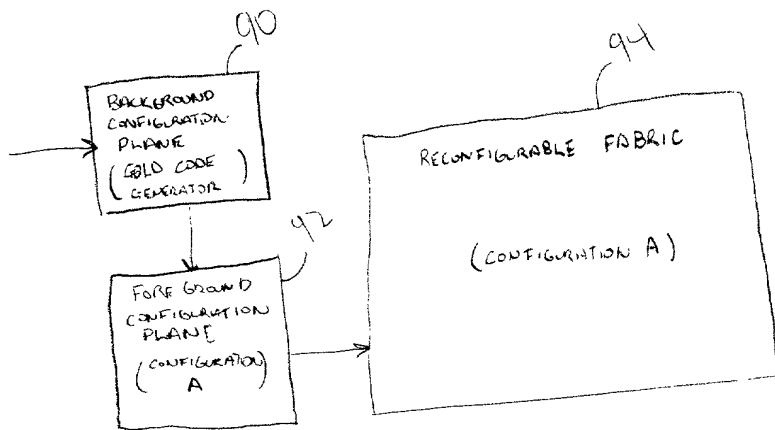


FIGURE 6A

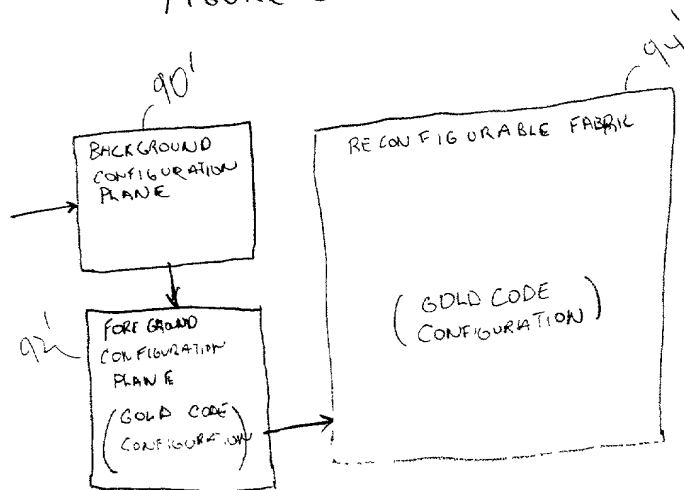


FIGURE 6B

$$C_{\text{long1},n} = \text{LSFRA}[7:0] \text{ XOR } \text{LSFRB}[7:0]$$

Let us define $\text{LFSRC}'[i] = \text{LSFRC}[2\lfloor i/2 \rfloor]$

$$C_{\text{long},n}(i) = C_{\text{long1},n}(i)(1 + j(-1)^i(c_{\text{long2},n}(2\lfloor i/2 \rfloor)) \text{ (from 3G TS25.213)}$$

Multiplying bits by +1/-1 is the same as XOR for 0s and 1s.
XORing by 0xAA can be used in place of the $(-1)^i$ term.

In binary representation, the Scrambling Code $C_{\text{long},n}$ becomes:

$$\begin{aligned} C_{\text{long},n}[7:0] &= C_{\text{long1},n}[7:0](1 + j(0xAA) \text{ XOR } C'_{\text{long2},n}[7:0]) \\ C_{\text{long},n}[7:0] &= \text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0] \\ &\quad + j(\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0] \text{ XOR } 0xAA \text{ XOR } \text{LFSRC}'[7:0] \text{ XOR } \text{LFSRD}'[7:0]) \\ C_{\text{long},n}[7:0] &= \text{SCI}[7:0] + \text{jSCQ}[7:0] \end{aligned}$$

Let us define $\text{LFSRD}''[7:0] = 0xAA \text{ XOR } \text{LFSRD}'[7:0]$, then:

$$\begin{aligned} C_{\text{long},n}[7:0] &= (\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0]) \\ &\quad + j(\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0] \text{ XOR } \text{LFSRC}'[7:0] \text{ XOR } \text{LFSRD}''[7:0]) \end{aligned}$$

We use a lookup table to compute $\text{LFSRC}'[7:0]$ and $\text{LFSRD}''[7:0]$

FIGURE 7

Gold Code Generator

Lookup[6:0] Definitions

<p>At Address 4n+0: OUT[7:0] = Next StateA[3:0]:PASSA[3:0]</p> <p>OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]</p>	<p>At Address 4n+2: OUT[7:0] = Next StateC[3:0]:LFSRC'[3:0]</p> <p>OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]</p>
<p>At Address 4n+1: OUT[7:0] = Next StateB[3:0]:PASSB[3:0]</p> <p>OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]</p>	<p>At Address 4n+3: OUT[7:0] = Next StateD[3:0]:LFSRD"[3:0]</p> <p>OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]</p>

FIGURE 8A

Gold Code Generator Lookup[10:4] Definitions

<p>At Address $4n+0$: $\text{OUT}[7:0] = \text{IN}[7:4] \cdot \text{Next StateA}[7:4]$</p> <p>$\text{OUT}[7] = \text{IN}[3]$ $\text{OUT}[6] = \text{IN}[2]$ $\text{OUT}[5] = \text{IN}[1]$ $\text{OUT}[4] = \text{IN}[0]$ $\text{OUT}[3] = \text{IN}[6] \text{ XOR } \text{IN}[3]$ $\text{OUT}[2] = \text{IN}[5] \text{ XOR } \text{IN}[2]$ $\text{OUT}[1] = \text{IN}[4] \text{ XOR } \text{IN}[1]$ $\text{OUT}[0] = \text{IN}[3] \text{ XOR } \text{IN}[0]$</p>	<p>At Address $4n+2$: $\text{OUT}[7:0] = \text{IN}'[7:4] \cdot \text{Next StateC}[7:4]$</p> <p>$\text{OUT}[3] = \text{IN}[2]$ $\text{OUT}[2] = \text{IN}[2]$ $\text{OUT}[1] = \text{IN}[0]$ $\text{OUT}[0] = \text{IN}[0]$ $\text{OUT}[7] = \text{IN}[6] \text{ XOR } \text{IN}[3]$ $\text{OUT}[6] = \text{IN}[5] \text{ XOR } \text{IN}[2]$ $\text{OUT}[5] = \text{IN}[4] \text{ XOR } \text{IN}[1]$ $\text{OUT}[4] = \text{IN}[3] \text{ XOR } \text{IN}[0]$</p>
<p>At Address $4n+1$: $\text{OUT}[7:0] = \text{IN}[7:4] \cdot \text{Next StateB}[7:4]$</p> <p>$\text{OUT}[7] = \text{IN}[3]$ $\text{OUT}[6] = \text{IN}[2]$ $\text{OUT}[5] = \text{IN}[1]$ $\text{OUT}[4] = \text{IN}[0]$ $\text{OUT}[3] = \text{IN}[6] \text{ XOR } \text{IN}[4] \text{ XOR } \text{IN}[3]$ $\text{OUT}[2] = \text{IN}[5] \text{ XOR } \text{IN}[2]$ $\text{OUT}[1] = \text{IN}[4] \text{ XOR } \text{IN}[1]$ $\text{OUT}[0] = \text{IN}[3] \text{ XOR } \text{IN}[0]$</p>	<p>At Address $4n+3$: $\text{OUT}[7:0] = \text{IN}''[7:4] \cdot \text{Next StateD}[7:4]$</p> <p>$\text{OUT}[3] = \text{IN}[2]$ $\text{OUT}[2] = \text{IN}[2]$ $\text{OUT}[1] = \text{IN}[0]$ $\text{OUT}[0] = \text{IN}[0]$ $\text{OUT}[7] = \text{IN}[6] \text{ XOR } \text{IN}[5] \text{ XOR } \text{IN}[4] \text{ XOR } \text{IN}[3]$ $\text{OUT}[6] = \text{IN}[5] \text{ XOR } \text{IN}[4] \text{ XOR } \text{IN}[3] \text{ XOR } \text{IN}[2]$ $\text{OUT}[5] = \text{IN}[4] \text{ XOR } \text{IN}[3] \text{ XOR } \text{IN}[2] \text{ XOR } \text{IN}[1]$ $\text{OUT}[4] = \text{IN}[3] \text{ XOR } \text{IN}[2] \text{ XOR } \text{IN}[1] \text{ XOR } \text{IN}[0]$</p>

FIGURE 8B

Chameleon Systems UMTS Rake Receiver Mapping Analysis

27-April-2000

Revision 0.41

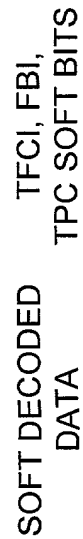
Dan Pugh & Mark Rollins
Chameleon Systems, Inc.
dan@chameleonsystems.com
rollins@chameleonsystems.com

Chameleon Systems Confidential

Chameleon Systems Confidential

Rake Receiver Requirements and Assumptions

- Requirements
 - ◆ Implement 32 users Pilot Processing and Data Despreading
 - ✦ Strive for target of 32 users in CS2112 (125 MHz)
 - ◆ Exceed target of 75 users in CS2112X (250 MHz)
- Assumptions
 - ◆ Maximum of 12 antennas
 - ◆ Maximum of 8 fingers per user
 - ◆ Average of 4 fingers per user
 - ◆ Spreading factors of 4 to 256 on DPDCH
 - ◆ Dual-port RAM at the input; Input order may be specified
 - ◆ ARC supplies scrambling code seeds
 - ◆ The Chameleon Processor is running at exactly 32 x the chip rate
 - ◆ An external Phase-locked-loop generates the 32 x (122.88) processor clock and is locked to the 3.84 MHz chip clock

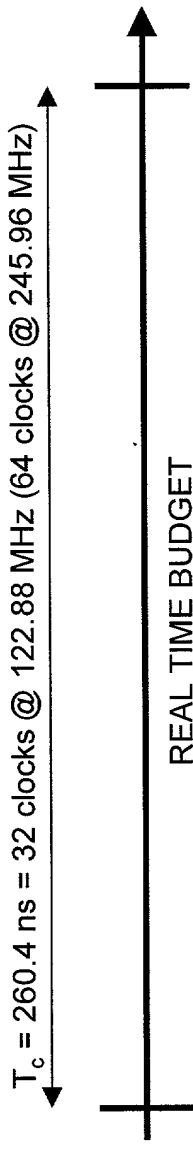
[illegible]

Rake Receiver

Description of Pipelined Operation

- Store a window of 128 chip samples sampled $T_c/2$ apart
- In each grouping of 256 consecutive 122.88 MHz clocks:
 - At EVERY 122.88 MHz clock period:
 - ◆ Read 8 consecutive chip samples at the correct multi-path delay offset
 - ◆ Align the eight samples to the Despreading Code (Gold Code)
 - ◆ Multiply the 8 data samples by the 8 appropriate Despreading Codes
 - ◆ Sum the eight despread chips into one sample (two samples if SF=4)
 - ◆ Accumulate the sum-of-eight chips into a single despread symbol
- Send the despread Pilot Symbols to the ARC processor
 - ◆ Calculate Channel Estimation Weights
 - ◆ Multiply TFCI, FBI, TPC bits by Channel Estimation Weights
 - ◆ Sum up to six fingers to form each soft bit (symbol)

Fundamental Scheduling Analysis



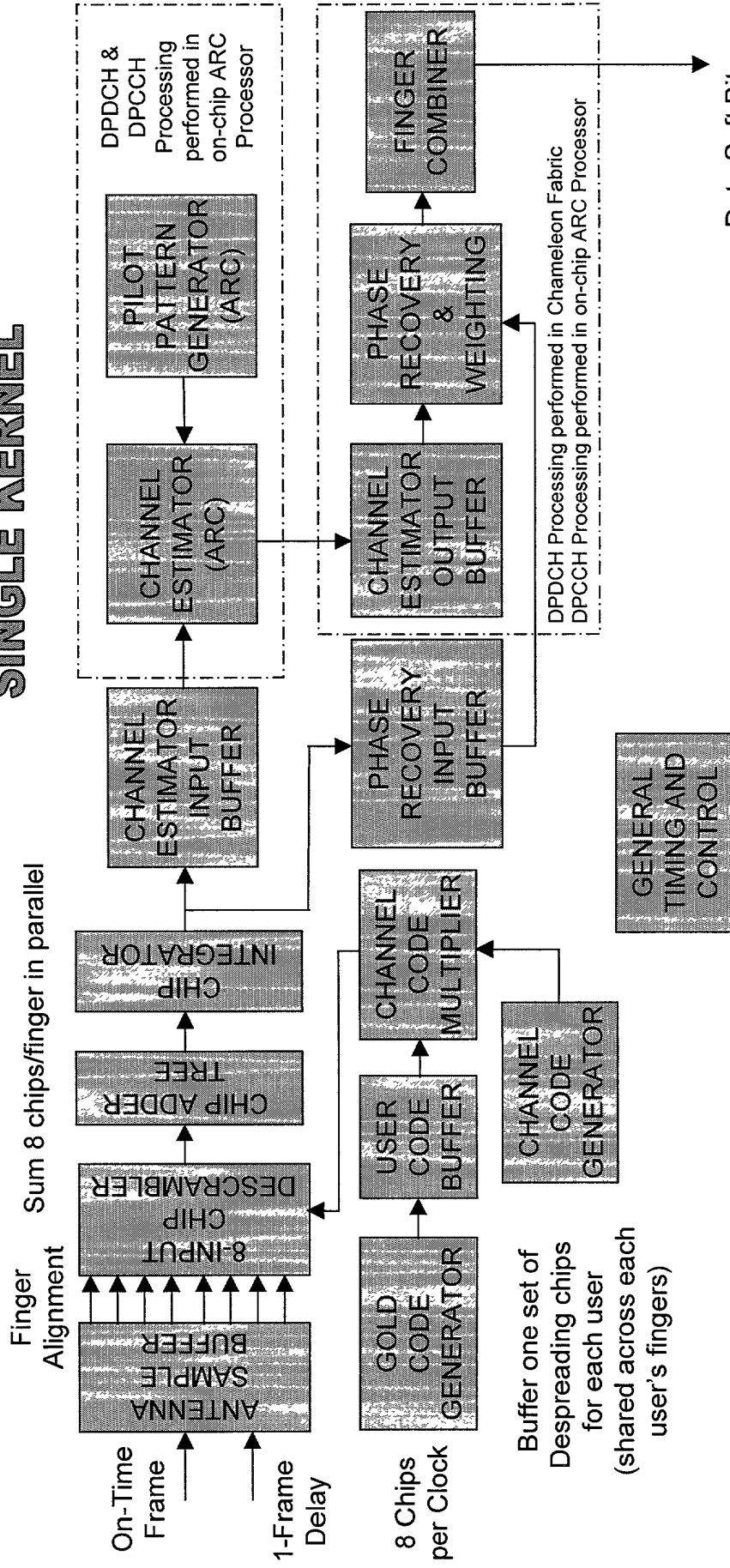
Parallel Implementation:

- Each circuit processes 8 chips per clock
- Throughput of 256 fingers per circuit
- Populate device with single circuit
- Achieves 128 pilot data fingers (32 users) @ 122.88 MHz
- Achieves 128 data fingers (same 32 users) @ 122.88 MHz
- Achieves 512 fingers (64 users) @ 245.96 MHz
- Utilizes single centralized control unit
- Fits within a single Chameleon device

Parallel Implementation is more efficient
than multiple instantiations of individual units

Chameleon Rake Processor High-Level Partitioning Overview

SINGLE CIRCUIT SINGLE KERNEL

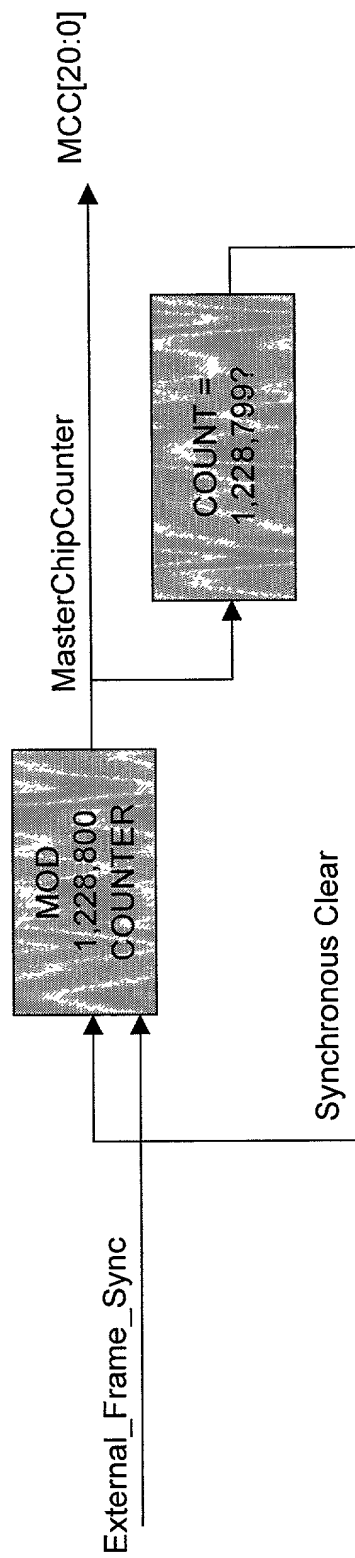


General Timing and Control

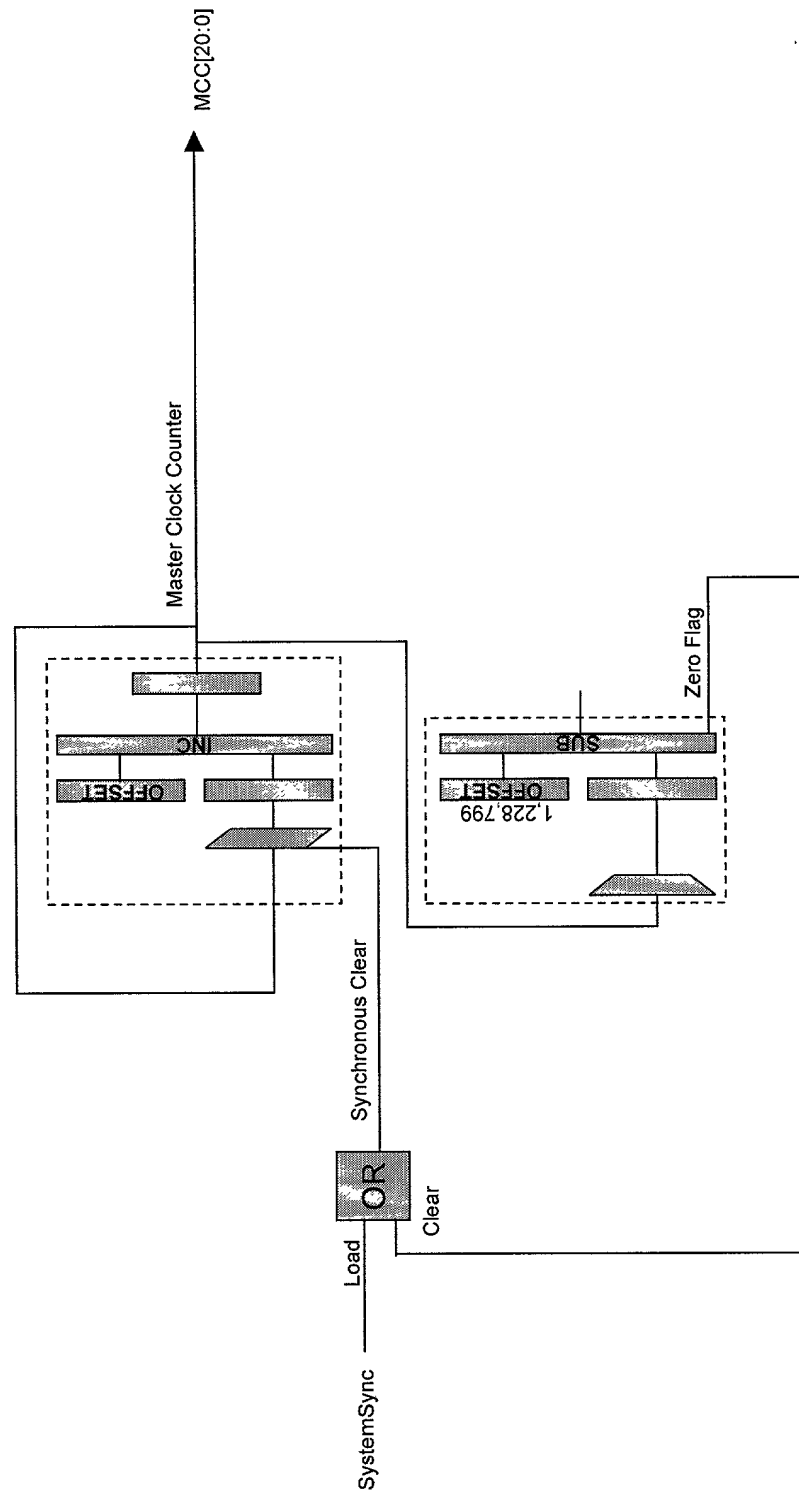
- The Rake Processor is synchronized to the chip-rate clock with an external Phase-Locked-Loop (PLL)
- The Rake Processor clock runs at exactly 32 times the chip-rate clock ($32 \times 3.84 \text{ MHz} = 122.88 \text{ MHz}$)
- All processes in the Rake Processor are synchronized to the Master Chip Counter (MCC)
- The MCC counts modulo 1,228,800 (32 clocks/chip x 15 slots x 2560 chips/slot) and is reset by the external Frame Sync signal
- Various bits in the MCC are used to generate the Antenna Sample Buffer Write Address

Master Chip Counter Block Diagram

The External_Frame_Sync signal clears the modulo 1,228,800 counter



Master Chip Counter Implementation



Master Chip Counter Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 2 DPUs
 - ◆ 0 LSMs

Antenna Sample Buffer Requirements and Assumptions

- Requirements
 - ◆ Provide memory in the Antenna Buffer so that a 128-chip (256 half-chips) range of samples can be stored for all 12 antennas
 - ◆ This allows the Finger Alignment Buffer and the Antenna Buffer to be merged into a single memory
- Assumptions
 - ◆ The samples must be organized such that any eight samples T_c apart from one antenna may be read simultaneously from the buffer
 - ◆ Each Rake Receiver circuit (Chameleon chip) processor services from one to six sectors
 - ◆ Each Sector must process the primary and diversity antennas for the primary sector and the two adjacent sectors, for a minimum of six antennas per sector of coverage
 - ◆ Support of all 12 antennas is required

Antenna Sample Buffer General Description

- There are two partitions of the Antenna Sample Buffer
 - ◆ The first partition contains the on-time antenna data
 - ◆ The second partition contains the on-time antenna data that is delayed by approximately one frame
- The value of the delay is one frame plus the time required to compute the Spreading Factor from the TFCI bits
- The delayed data is required because the TFCI bits are in the same frame as the data that it controls

Antenna Sample Buffer General Description

- The Antenna Sample Buffer is used to store a large enough window of data that all six fingers may be accessed from any of the 12 antennas
- Data for each finger is read from the Antenna Buffer at the offset specified by the Path Searcher
- The Antenna Buffer is organized so that ANY eight consecutive samples T_c apart may be accessed from the buffer in a single clock cycle

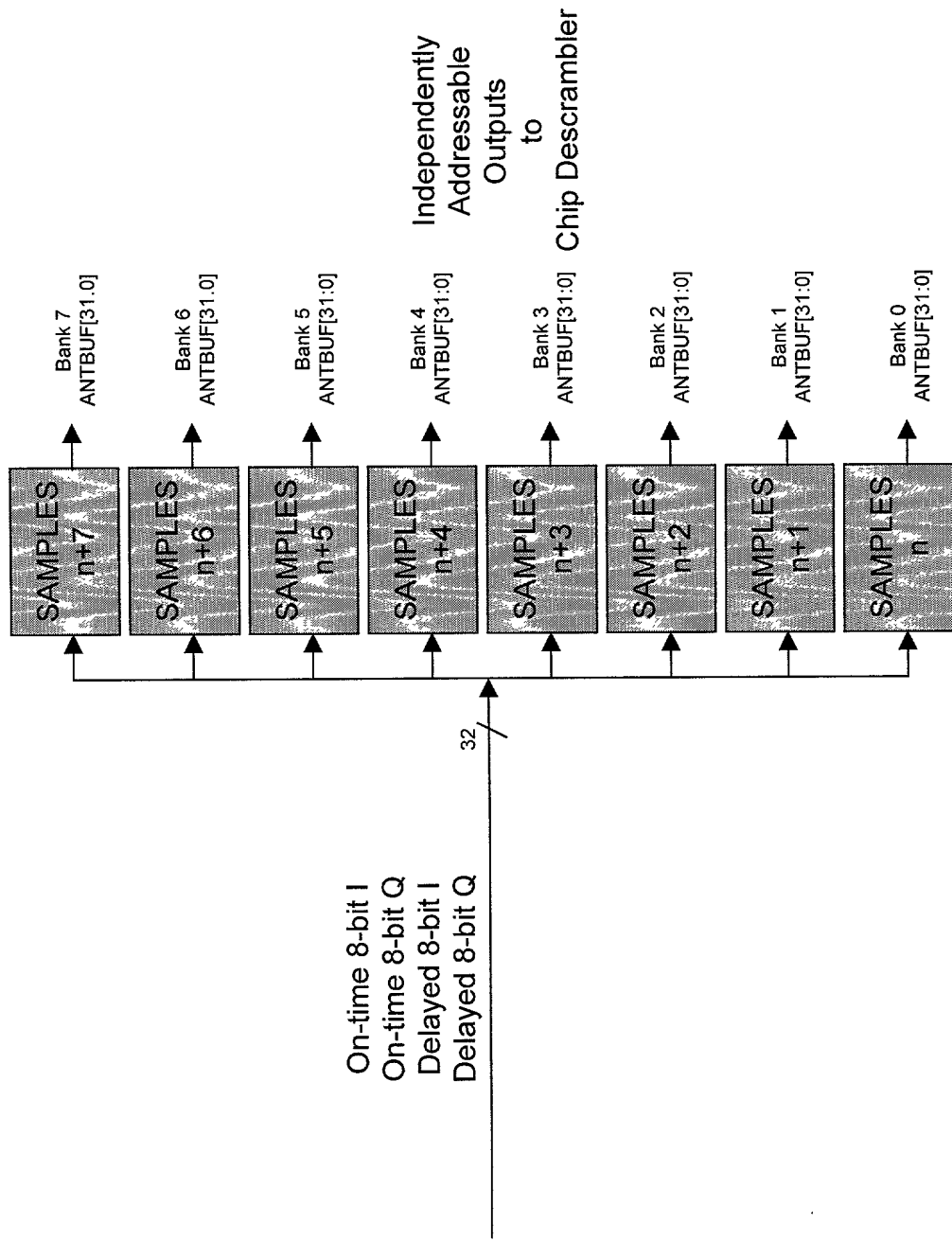
Antenna Sample Buffer Multipath Support Capabilities

- Specifications:
 - ◆ Chip-rate = 3.84 MHz (260.4 ns, wavelength = 78.125 m)
 - ◆ Maximum distance of mobile user to base station = 7000 m
- The 7000 m user radius corresponds 0-89.6 chips of line-of-sight delay
- The 128-chip Antenna Sample Buffer provides an additional 38.4-chip buffer to support multipath components
- This corresponds to support for multipath components with up to 10,000 ns (3000 m) delay for all 12 antennas
- If the Antenna Sample Buffer is retasked to support only 8 antennas (two adjacent sectors) the maximum multipath support may be increased to 26,664 ns (8000 m)

Antenna Sample Buffer Multipath Support Capabilities

- Given a 5000m user radius (3G TS25.942)
 - ◆ A 5000 m user radius corresponds 64 chips of line-of-sight delay
 - ◆ The 128-chip Antenna Sample Buffer provides an additional 64-chip buffer to support multipath components
- This corresponds to 10,000m of support for a combination of the user radius plus multipath delay with all 12 antennas
- The Antenna Sample Buffer may be retasked to support:
 - ◆ 8 antennas (two sectors) for 15,000m of user radius/multipath support
 - ◆ 6 antennas (one sector) for 20,000m of user radius/multipath support

Antenna Sample Buffer Functional Block Diagram



Data Buffer

Chameleon Systems Confidential

Antenna Sample Buffer Memory Map for Bank 0

ADDRESS ANTBUF[31:0]

0x5FC	Antenna 11, Sample 120, HalfChip 1
0x5F8	Antenna 11, Sample 120, HalfChip 0
0x5F4	Antenna 11, Sample 112, HalfChip 1
0x5F0	Antenna 11, Sample 112, HalfChip 0

⋮

0x094	Antenna 1, Sample 16, HalfChip 1
0x090	Antenna 1, Sample 16, HalfChip 0
0x08C	Antenna 1, Sample 8, HalfChip 1
0x088	Antenna 1, Sample 8, HalfChip 0
0x084	Antenna 1, Sample 0, HalfChip 1
0x080	Antenna 1, Sample 0, HalfChip 0
0x07C	Antenna 0, Sample 120, HalfChip 1
0x078	Antenna 0, Sample 120, HalfChip 0
0x074	Antenna 0, Sample 112, HalfChip 1
0x070	Antenna 0, Sample 112, HalfChip 0

⋮

0x014	Antenna 0, Sample 16, HalfChip 1
0x010	Antenna 0, Sample 16, HalfChip 0
0x00C	Antenna 0, Sample 8, HalfChip 1
0x008	Antenna 0, Sample 8, HalfChip 0
0x004	Antenna 0, Sample 0, HalfChip 1
0x000	Antenna 0, Sample 0, HalfChip 0

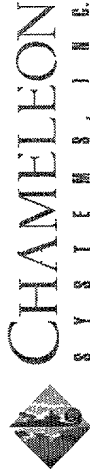
ANTENNA SAMPLE BUFFER WORD CONTENTS

ANTBUF[31:24]	ANTBUF[23:16]	ANTBUF[15:8]	ANTBUF[7:0]
Non-Delayed Q[7:0]	Delayed Q[7:0]	Non-Delayed I[7:0]	Delayed I[7:0]

Note:

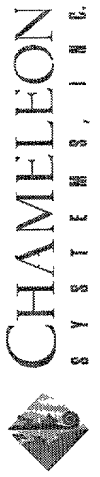
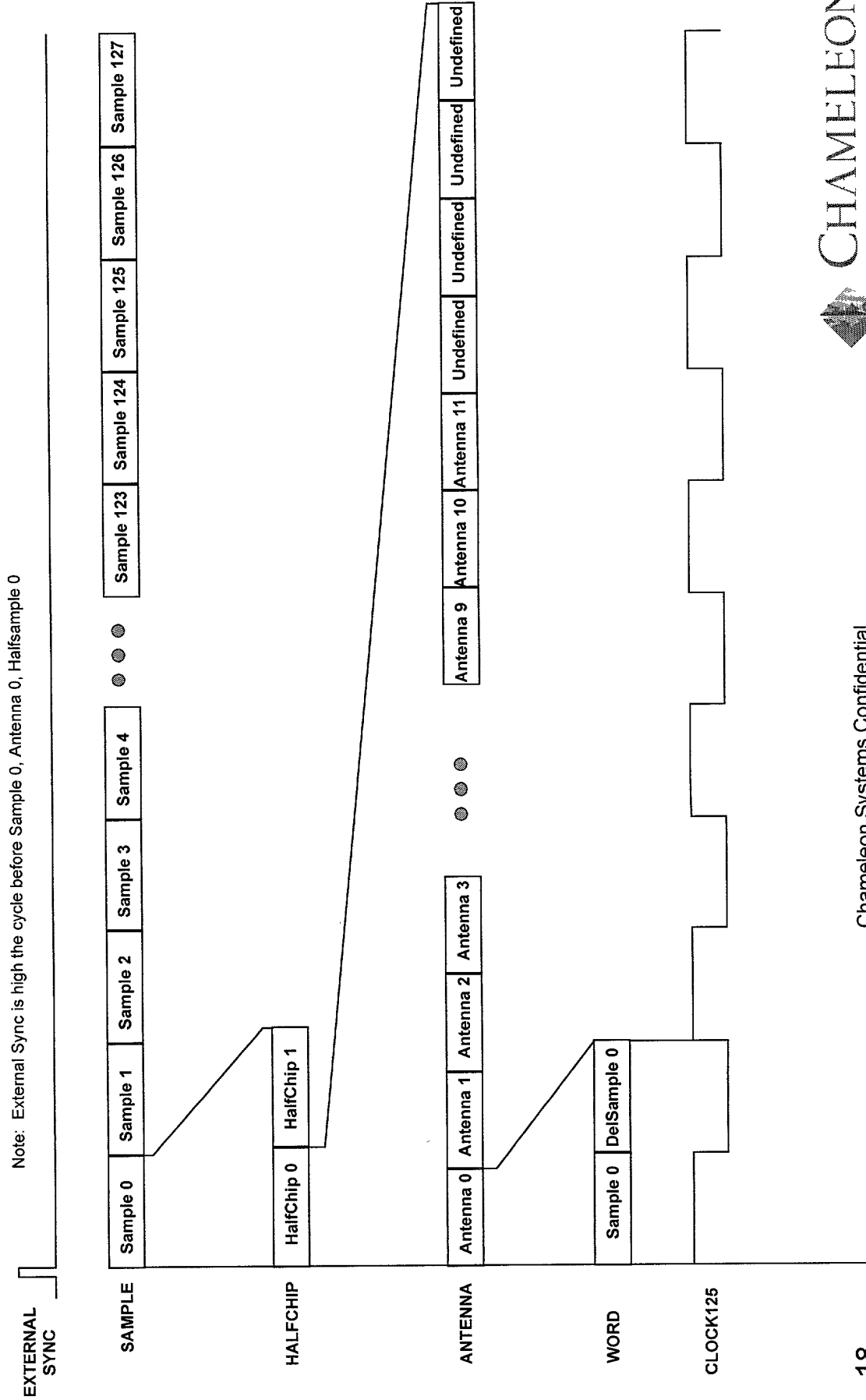
Sample n HalfChip 0 is used to denote the sample at time n

Sample n HalfChip 1 is used to denote the sample at time n+1/2

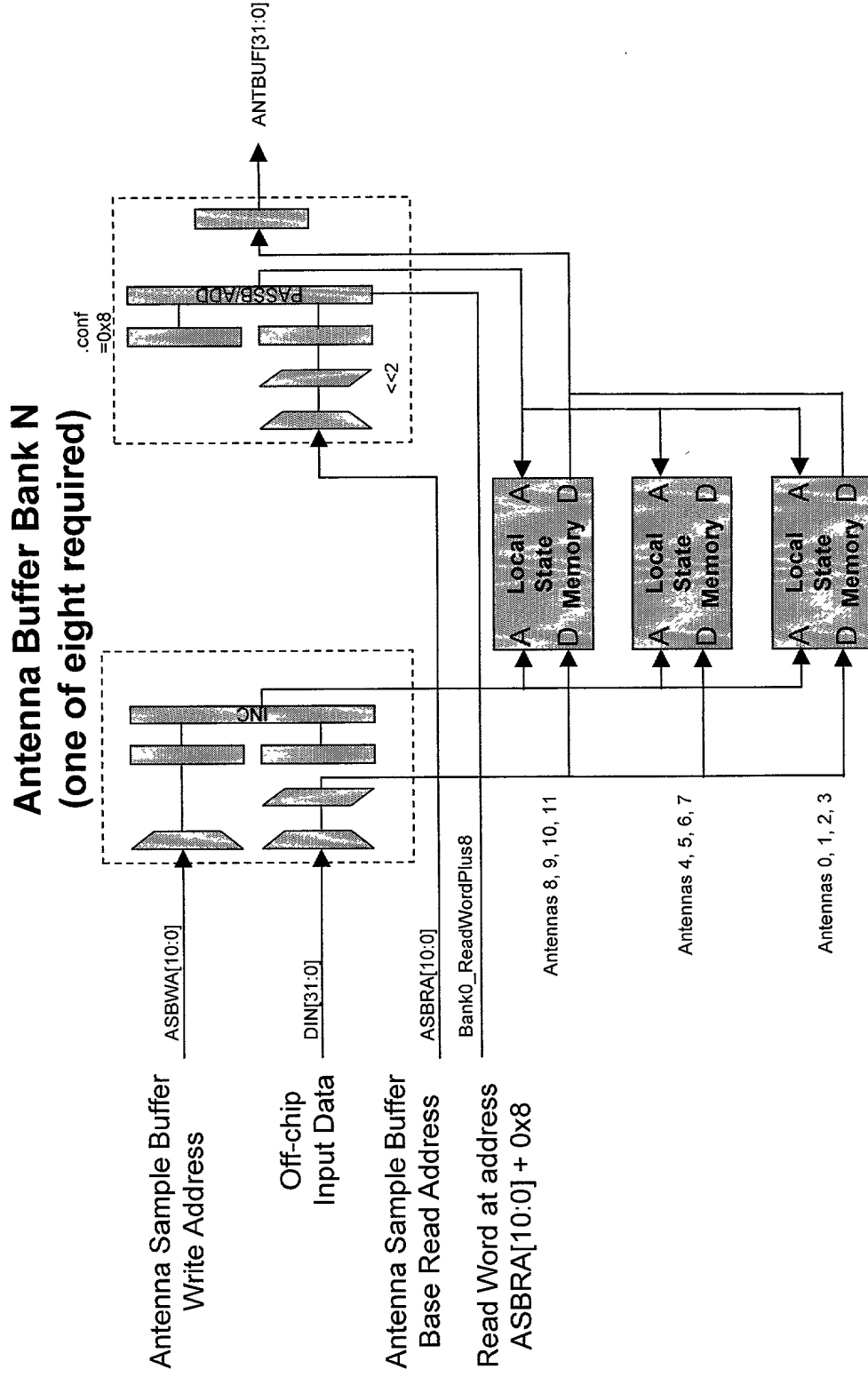


Chameleon Systems Confidential

External Antenna Sample Buffer Bus Organization



Antenna Sample Buffer Data Buffer Implementation



Antenna Sample Buffer Write Address Bit Definitions

- The Modulo 1,228,00 Master Chip Counter (MCC) bit fields may be defined with respect to the input data samples to the Antenna Sample Buffer Write Address (ASBWA)
- The Antenna Sample Buffer Write Address Generator bits are MCC bits that have been reordered to properly store the input samples

**ANTENNA SAMPLE BUFFER
BANK ADDRESS**
(for each of the eight banks)

MCC BIT	DESCRIPTION
MCC[11]	ChipCount[6]
MCC[10]	ChipCount[5]
MCC[9]	ChipCount[4]
MCC[8]	ChipCount[3]
MCC[7]	ChipCount[2]
MCC[6]	ChipCount[1]
MCC[5]	ChipCount[0]
MCC[4]	HalfChip
MCC[3]	Antenna[3]
MCC[2]	Antenna[2]
MCC[1]	Antenna[1]
MCC[0]	Antenna[0]

ASBWA BIT	DESCRIPTION
ASBWA[10]	Antenna[3]
ASBWA[9]	Antenna[2]
ASBWA[8]	Antenna[1]
ASBWA[7]	Antenna[0]
ASBWA[6]	ChipCount[6]
ASBWA[5]	ChipCount[5]
ASBWA[4]	ChipCount[4]
ASBWA[3]	ChipCount[3]
ASBWA[2]	HalfChip
ASBWA[1]	0
ASBWA[0]	0

Note:

ChipCount[2:0] is used to select one of eight physical memory banks

The one-frame-delayed

Samples are stored in bytes 0 and 2 of the Antenna Sample Buffer, while the non-delayed samples are stored in bytes 1 and 3 of the Antenna Sample Buffer

Each Bank requires 3 LSMs

Antenna Sample Buffer Write Address Generation

- If we look at the fields of the Write Address Generator we see that:
 - ◆ CASE A: When $MCC[11:0] = 0xFFFF$
 - ✦ $ASBWA[10:7] = Antenna[3:0]$ must be cleared
 - ✦ $ASBWA[6:3] = ChipCount[6:3]$ must be cleared
 - ✦ $ASBWA[2] = HalfChip$ must be cleared
 - ◆ CASE B: When $MCC[7:0] = 0xFF$
 - ✦ $ASBWA[10:7] = Antenna[3:0]$ must be cleared (by incrementing by one)
 - ✦ $ASBWA[6:3] = ChipCount[6:3]$ increments by one
 - ✦ $ASBWA[2] = HalfChip$ must be cleared
 - ◆ CASE C: When $MCC[4:0] = 0x1F$
 - ✦ $ASBWA[10:7] = Antenna[3:0]$ increments by one
 - ✦ $ASBWA[2] = HalfChip$ must be cleared
 - ◆ CASE D: When $MCC[4:0] = 0x0F$
 - ✦ $ASBWA[10:7] = Antenna[3:0]$ increments by one
 - ✦ $ASBWA[2] = HalfChip$ must be set
 - ◆ CASE E: Otherwise
 - ✦ $ASBWA[10:7] = Antenna[3:0]$ must be incremented by one

Antenna Sample Buffer Write Address Generation Implementation

- To implement the above five cases, let us define two registers:
 - ◆ $\text{REGA} = 128 - 4 = 124 = 0x7C$
 - ◆ $\text{REGB} = 128 = 0x80$
- CASE A: This state occurs when the entire Antenna Sample Buffer has been written and the address field must be cleared so that the buffer may start again at the beginning address
 - $\text{MCC}[11:0] = 0xFFFF$
 - ◆ $\text{ASBWA}[10:7] = \text{Antenna}[3:0]$ must be cleared
 - ◆ $\text{ASBWA}[6:3] = \text{ChipCount}[6:3]$ must be cleared
 - ◆ $\text{ASBWA}[2] = \text{HalfChip}$ must be cleared
 - ✦ $\text{ALUB} = \text{shifterconst} = 0x0$
 - ✦ $\text{ALU} = \text{PASSB}$

Antenna Sample Buffer Write Address Generation Implementation

- CASE B: This state occurs when both a chip and HalfChip have been written to each of the eight banks and the address field must point back to the first bank
- $MCC[7:0] = 0xFF$
 - ◆ $ASBWA[10:7] = Antenna[3:0]$ increments by one
 - ◆ $ASBWA[6:3] = ChipCount[6:3]$ increments by one
 - ◆ $ASBWA[2] = HalfChip$ must be cleared
 - ✦ Since we know $ChipCount[6:3] \neq 0xFF$, incrementing $ChipCount[6:3]$ will not generate a carry into $Antenna[3:0]$
 - ✦ Since we know $HalfChip = 1$, we can increment HalfChip and it will toggle HalfChip AND generate a carry to increment $ChipCount[6:3]$
 - ✦ $ALUA = ALUOUTREG$
 - ✦ $ALUB = 128 + 4 = REGB \text{ OR } shifterconst = 0x4$
 - ✦ $ALU = ALUA + ALUB$

Antenna Sample Buffer Write Address Generation Implementation

- CASE C: This state occurs when after a sample (HalfChip=1) has been written to a single bank for all twelve antennas and the address field must point to the next memory bank
- $MCC[4:0] = 0x1F$
 - ◆ $ASBWA[10:7] = Antenna[3:0]$ increments by one
 - ◆ $ASBWA[2] = HalfChip$ must be cleared
 - ✦ $ALUA = 128 - 4 = REGA$
 - ✦ $ALUB = ALUOUTREG$
 - ✦ $ALU = ALUA + ALUB$

Antenna Sample Buffer Write Address Generation Implementation

- CASE D: This state occurs when after a sample (HalfChip=0) has been written to a single bank for all twelve antennas and the address field must point to the next half-chip address within the same memory bank
- $MCC[4:0] = 0x0F$
 - ◆ $ASBWA[10:7] = Antenna[3:0]$ increments by one
 - ◆ $ASBWA[2] = HalfChip$ must be set
 - ✦ $ALUA = ALUOUTREG$
 - ✦ $ALUB = 128 + 4 = REGB \text{ OR } shifterconst = 0x4$
 - ✦ $ALU = ALUA + ALUB$

Antenna Sample Buffer Write Address Generation Implementation

- CASE E: This state occurs when the conditions for any of the cases A through D are not met and the address field must point to the next antenna sample
 - ◆ $\text{ASBWA}[10:7] = \text{Antenna}[3:0]$ must be incremented by one
 - ✦ $\text{ALUA} = \text{ALUOUTREG}$
 - ✦ $\text{ALUB} = 128 = \text{REGB}$
 - ✦ $\text{ALU} = \text{ALUA} + \text{ALUB}$
- Note that for both CASE B and CASE D, the instructions to the DPU are identical
- We therefore only have four unique states for the DPU

Antenna Sample Buffer Write Address Generation Implementation

- Let us define the following four states
 - ◆ State 00: DPU instruction for CASE A
 - ◆ State 01: DPU instruction for CASE B and CASE D
 - ◆ State 10: DPU instruction for CASE C
 - ◆ State 11: DPU instruction for CASE E
- Let us use the notation of A when case A is true and /A when CASE A is not active, and X represents don't care
- We also must assign priority to the five cases since the priority is base on the MCC[11:0] higher order bits having higher priority

◆ Priority 1: A , CASE B=X, C=X, D=X E=X	State 00
◆ Priority 2: (!A & B), C=X, D=X E=X	State 01
◆ Priority 3: (!A & !B & C & !D)	State 10
◆ Priority 3: (!A & !B & !C & D)	State 01
◆ Note CASE C and CASE D have equal priority	
◆ Priority 4: !A & !B & !C & !D	State 11
◆ Note that the encoding !A & !B & C & D is not physically possible	

Antenna Sample Buffer Write Address Generation Implementation

- Let P_n represent the priority for priority with level n
- We can fill in the priority assignments for the Karnaugh map entries for the sixteen possibilities for P_0 , P_1 , P_2 , and P_3

PRIORITY ASSIGNMENTS

		$\begin{matrix} D=1 & C=1 \\ \hline \end{matrix}$			
		00	01	11	10
$\begin{matrix} B=1 \\ \hline A=1 \end{matrix}$	00	P4	P3	X	P3
	01	P2	P2	P2	P2
	11	P1	P1	P1	P1
	10	P1	P1	P1	P1

Antenna Sample Buffer Write Address Generation Implementation

If we fill in the DPU state tables with the state assignments of the previous page:

Priority 1: A, CASE B=X, C=X, D=X E=X State 00

Priority 2: (!A & B), C=X, D=X E=X State 01

Priority 3: (!A & !B & C & !D) State 10

Priority 3: (!A & !B & !C & D) State 01

Note CASE C and CASE D have equal priority

Priority 4: !A & !B & !C & !D State 11

Note that the encoding !A & !B & C & D is not physically possible

STATE[1]					
		D=1		C=1	
		00	01	11	10
B=1	00	1	0	X	1
	01	0	0	0	0
	11	0	0	0	0
	10	0	0	0	0

STATE[0]					
		D=1		C=1	
		00	01	11	10
B=1	00	1	1	X	0
	01	1	1	1	1
	11	0	0	0	0
	10	0	0	0	0

		D=1		C=1	
		00	01	11	10
B=1	00	1	0	X	1
	01	0	0	0	0
	11	0	0	0	0
	10	0	0	0	0

		D=1		C=1	
		00	01	11	10
B=1	00	1	1	X	0
	01	1	1	1	1
	11	0	0	0	0
	10	0	0	0	0

STATE[1] = !A & !B & !D

STATE[0] = (!A & B) | (!B & !C)

Note that we do not need to compute variable D

Antenna Sample Buffer Write Address Generation Implementation

In order to be able to decode the conditions A, B, C, and D and their inverses, without using excessive product terms, we decode a count one before the desired count and register the result

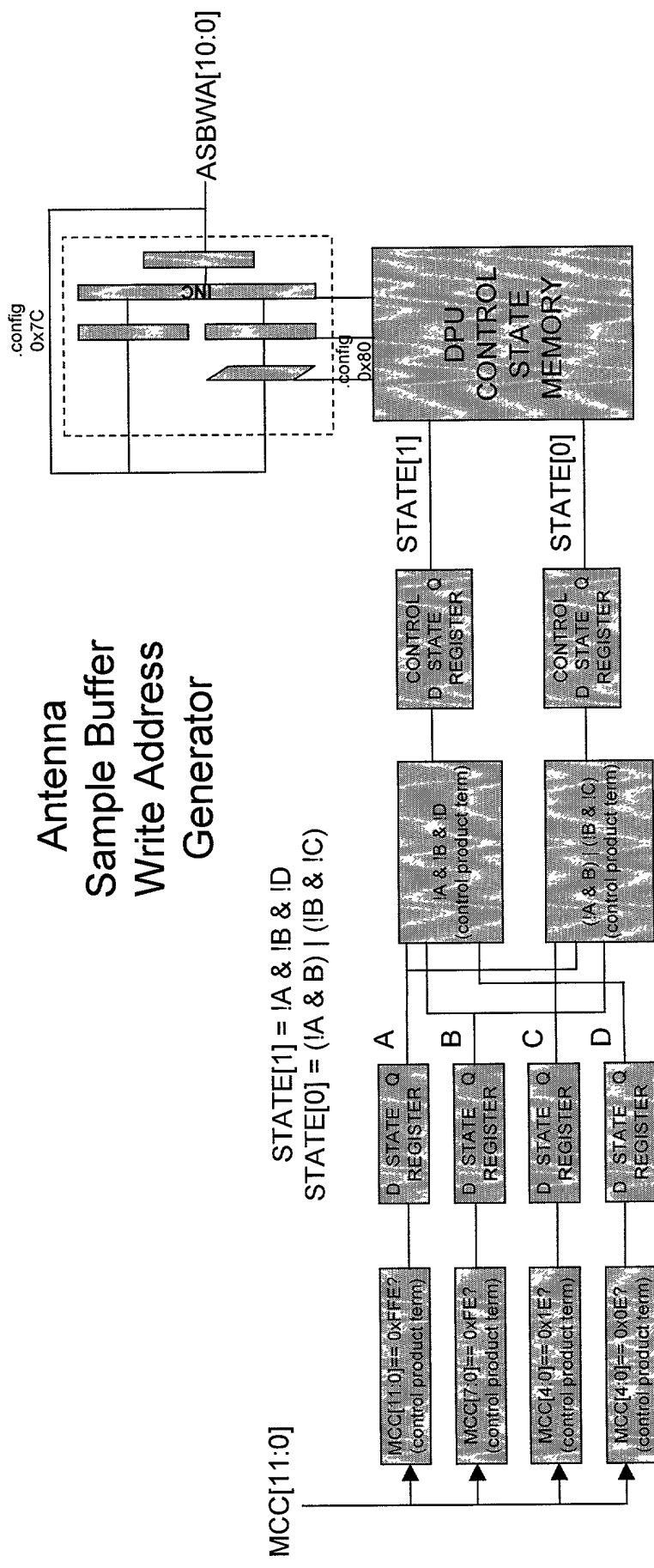
REGA: $MCC[11:0] = 0xFFE = MCC[11] \& MCC[10] \& MCC[9] \& MCC[8] \& MCC[7] \& MCC[6] \& MCC[5] \& MCC[4] \& MCC[3] \& MCC[2] \& MCC[1] \& !MCC[0]$

REGB: $MCC[7:0] = 0xFE = MCC[7] \& MCC[6] \& MCC[5] \& MCC[4] \& MCC[3] \& MCC[2] \& MCC[1] \& !MCC[0]$

REGC: $MCC[4:0] = 0x1E = MCC[4] \& MCC[3] \& MCC[2] \& MCC[1] \& !MCC[0]$

REGD: $MCC[4:0] = 0x0E = !MCC[4] \& MCC[3] \& MCC[2] \& MCC[1] \& !MCC[0]$

Antenna Sample Buffer Write Address Generator



Antenna Sample Buffer Bank Write Enable Generation

- Each of the eight banks in the Antenna Sample Buffer contains the samples eight chips apart for all twelve antennas.
- The control must generate a Antenna Sample Buffer Write Enable ($ASBWE_n$) signal for each of the n banks
- The timing for the $ASBWE_n$ signals is based upon the Master Chip Counter bits $MCC[7:0]$
- Note that the MCC counter implicitly counts through values for sixteen antennas, but the write enable signals are only enabled during the first twelve

Antenna Sample Buffer Bank Write Enable Generation

- ASBWE_n is active when MCC[7:5] = ChipCount[2:0]
matches the addressed memory bank and MCC[3:0] =
Antenna[3:0] addresses antennas 0 to 11:

- ◆ ASBWE₀ = IMCC[7] & IMCC[6] & IMCC[5] & i(MCC[3] & MCC[2])
ASBWE₁ = IMCC[7] & IMCC[6] & MCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₂ = IMCC[7] & MCC[6] & IMCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₃ = IMCC[7] & MCC[6] & MCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₄ = MCC[7] & IMCC[6] & IMCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₅ = MCC[7] & IMCC[6] & MCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₆ = MCC[7] & MCC[6] & IMCC[5] & i(MCC[3] & MCC[2])
- ◆ ASBWE₇ = MCC[7] & MCC[6] & MCC[5] & i(MCC[3] & MCC[2])

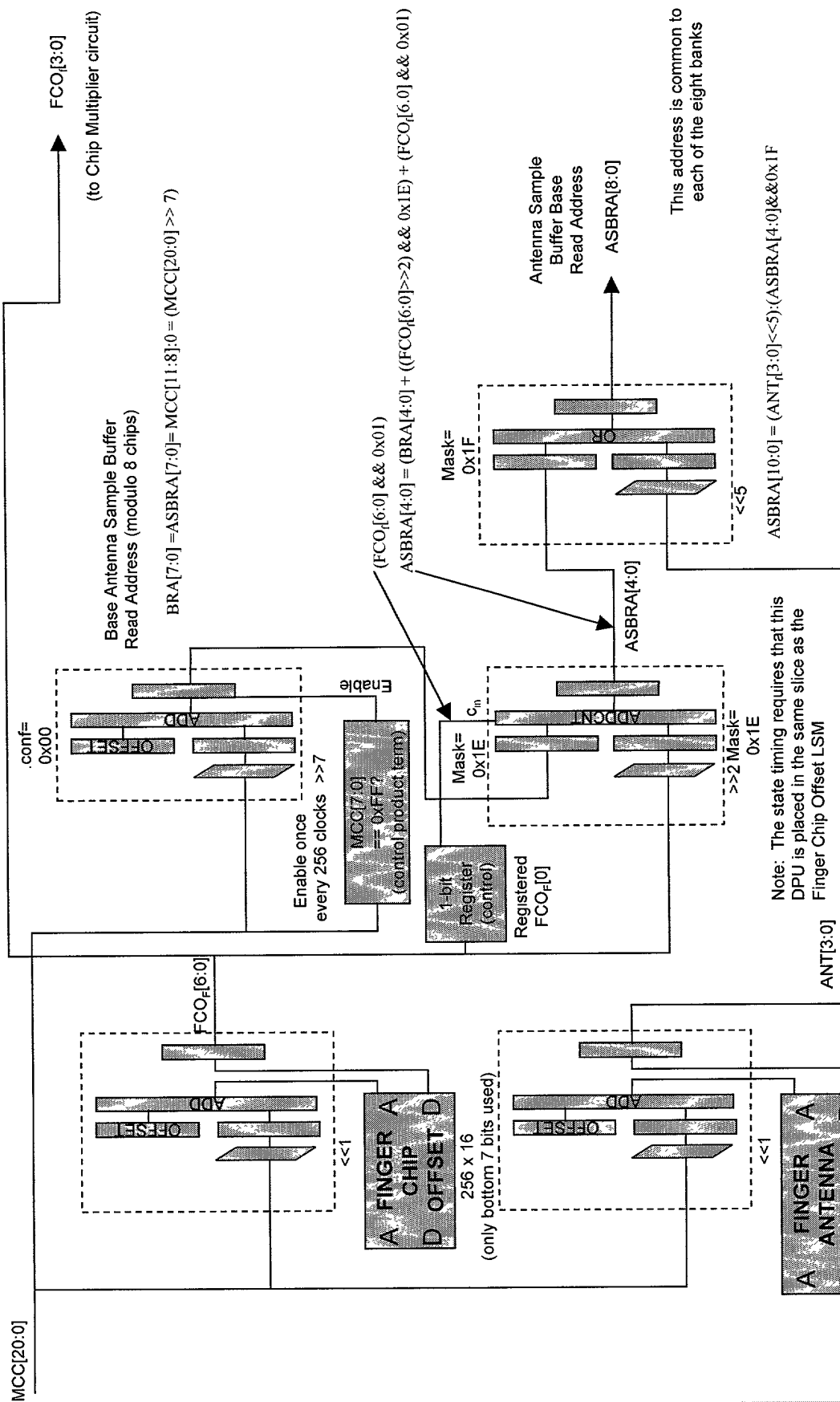
Antenna Sample Buffer Read Address Generation (1 of 2)

- Compute the Antenna Sample Buffer Base Read Address (ASBRA) for the Antenna Sample Buffer as follows:
 - ◆ The Base Read Address (BRA) is the fixed offset between the Antenna Sample Buffer Write Address and the Antenna Sample Buffer Read Address (ASRA) for zero finger chip offset
 - ◆ For each of the 256 fingers:
 - ✦ Add the Finger Chip Offset (FCO_f) to the BRA
 - ✦ Merge the Finger Antenna Assignment (ANT_f) bits to the ASBRA
 - ✦ The Antenna Sample Buffer Read Address (ASBA) is formed by shifting the ASBRA left two bit positions

Antenna Sample Buffer Read Address Generation (2 of 2)

- ◆ Compute the Base Read Address (BRA):
 - ✦ $BRA[4:0] = ASBRA[4:0] = MCC[11:8]:0 = ((MCC[20:0] \gg 7) \& 0x1E)$
 - The Base Read Address is latched to remain constant for 256 clocks
 - An offset can be added if necessary to adjust timing
- ◆ For each of the f fingers (0-255)
 - ✦ Read the antenna assignment (ANT_f) for the current finger
 - ✦ Read the Finger Chip Offset (FCO_f) for the current finger
 - ✦ Add the Finger Chip Offset to the Base Address Register
 - $ASBRA[4:0] = (BRA[4:0] + ((FCO_f[6:0] \gg 2) \& 0x1E) + (FCO_f[6:0] \& 0x01))$
 - ✦ The Antenna Assignment $ANT_f[3:0]$ is placed in bits $ASBRA[8:5]$
 - $ASBRA[8:0] = (ANT_f[3:0] \ll 5) : ASBRA[4:0]$
 - ✦ The Antenna Sample Buffer Read Address (ASRA) is formed by shifting $ASBRA$ left two bit positions
 - $ASRA[10:0] = ASBRA[8:0] \ll 2$

Antenna Sample Buffer Base Read Address Generator Implementation



Antenna Sample Buffer Read Address Offset Circuit

- The Antenna Sample Buffer Base Read Address, (BRA) is on a multiple of eight sample boundary
- If the Finger Chip Offset for a finger, FCO_f , is not on a multiple-of-eight sample boundary, then the correct eight consecutive samples are not output on the memory
- To output the correct words, some of the addresses must not select the word specified by the ASBRA, but the next full sample into the buffer
- $FCO_f[3:1]$ are used to determine which of the eight Antenna Sample Buffer address generators need to have their Bankn_ReadWordPlus8 asserted in order to read the next eighth sample into the buffer

Antenna Sample Buffer Read Address Offset Circuit

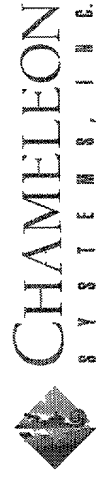
- The ASBRA calculation requires that the Finger Chip Offset for a given finger, FCO_f , is read out of the FCO memory before it is needed for the Read Address Offset Circuit
 - ◆ $FCO_f[0]$ is needed one cycle after it is read from the FCO memory, so the DPU using $FCO_f[0]$ as a control input must be placed in the same tile as the FCO memory, because $FCO_f[0]$ is delayed one cycle through the Control State Memory
 - ◆ $FCO_f[3:1]$ is needed by the Read Address Offset Circuit two cycles after it is read from the FCO memory
 - ◆ An additional delay of one cycle for $FCO_f[3:1]$ is achieved if $FCO_f[3:1]$ is routed through Broadcast Bit horizontal long lines and the following equations are used:

Antenna Sample Buffer Bankn_ReadWordPlus8

Antenna Sample Buffer Bankn_ReadWordPlus8 Generation

Case FCOff(3:1)

- 0: Bank0_ReadWordPlus8 = 0
Bank1_ReadWordPlus8 = 0
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 1: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 0
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 2: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 0
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 3: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 0
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 4: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 0
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 5: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 0
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 6: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 1
Bank6_ReadWordPlus8 = 0
Bank7_ReadWordPlus8 = 0
- 7: Bank0_ReadWordPlus8 = 1
Bank1_ReadWordPlus8 = 1
Bank2_ReadWordPlus8 = 1
Bank3_ReadWordPlus8 = 1
Bank4_ReadWordPlus8 = 1
Bank5_ReadWordPlus8 = 1
Bank6_ReadWordPlus8 = 1
Bank7_ReadWordPlus8 = 0



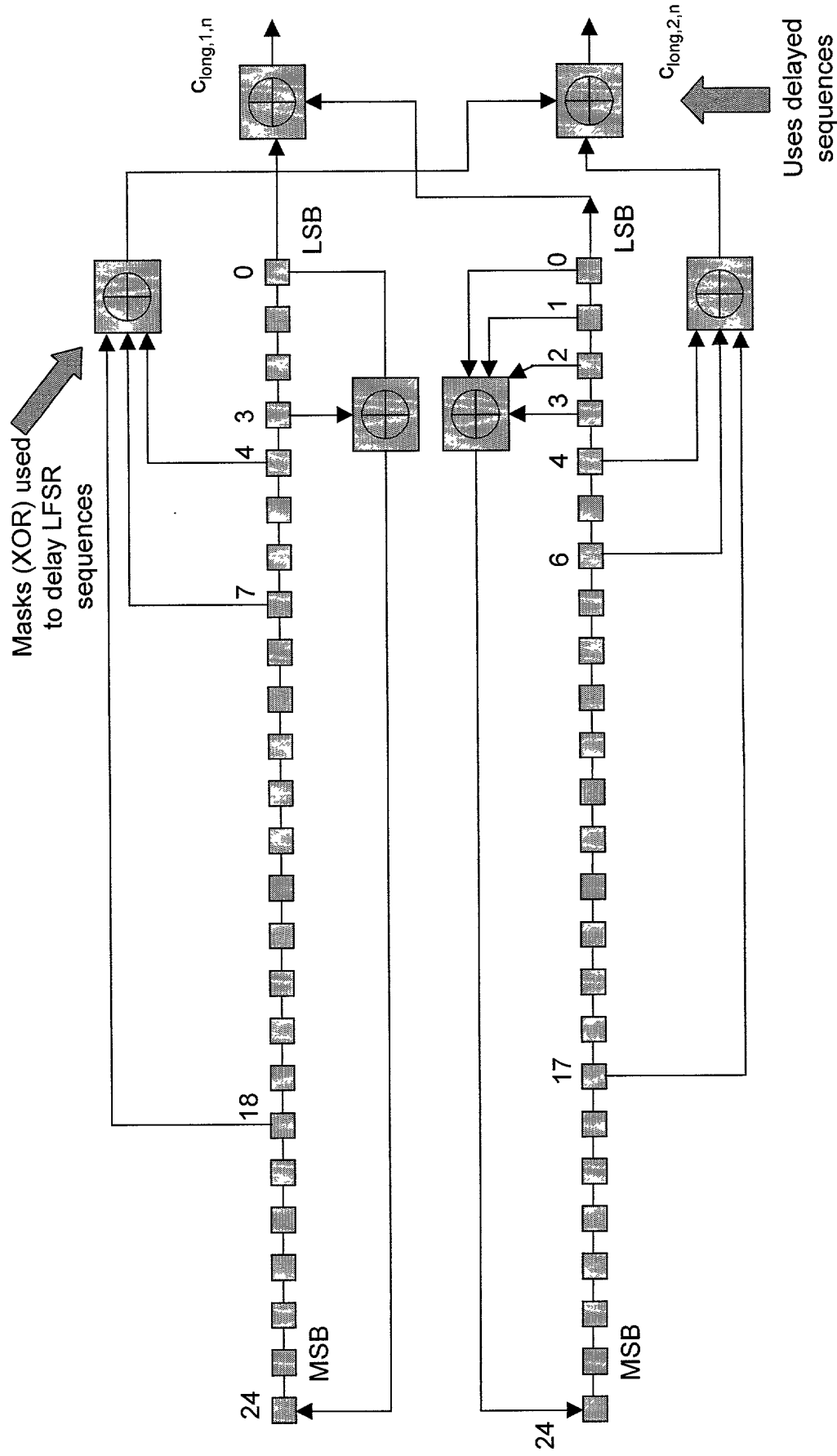
Antenna Sample Buffer Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 22 DPUs
 - ◆ 26 LSMs

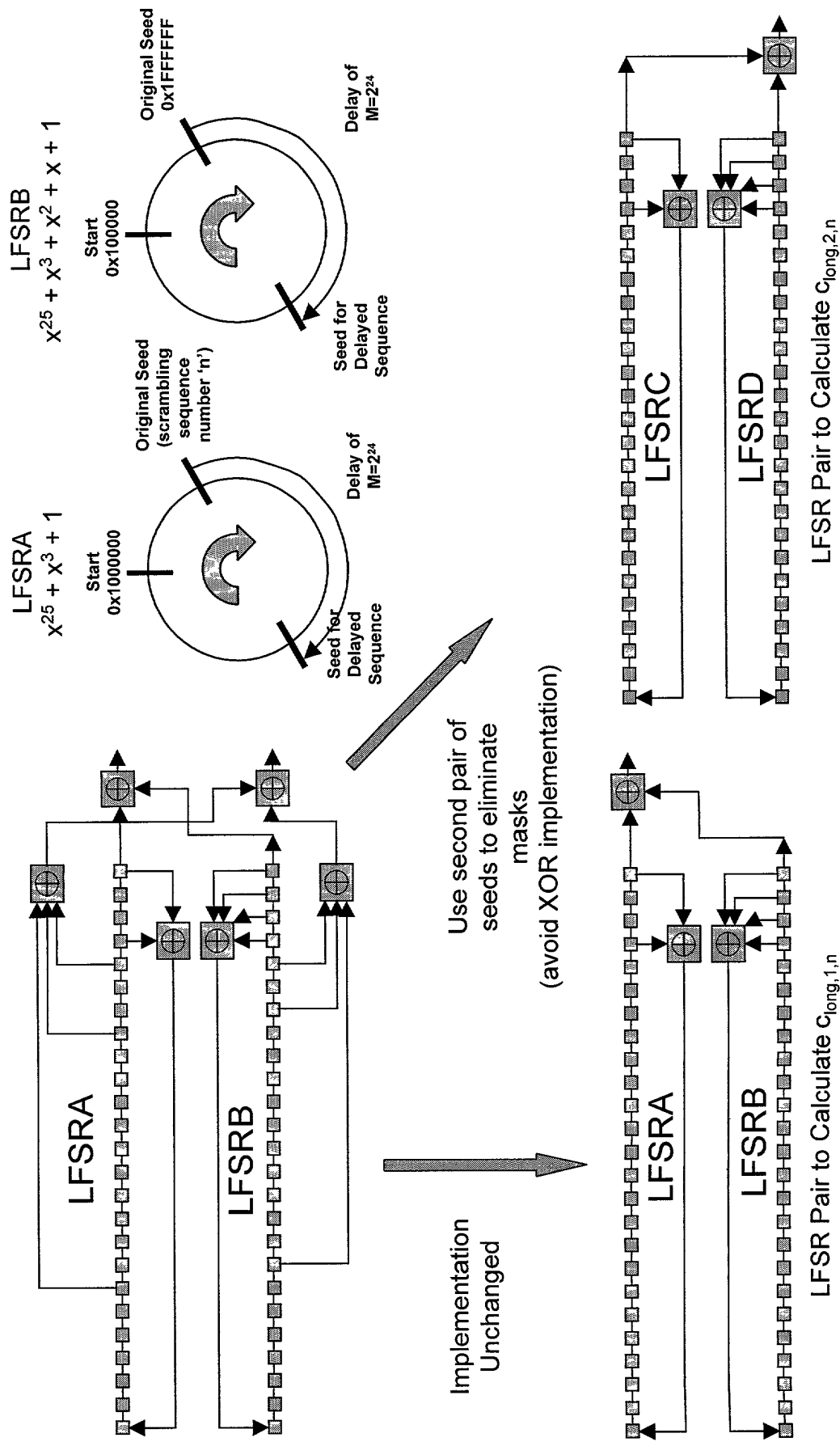
UMTS Gold Code Generator Requirements & Assumptions

- Requirements
 - ◆ Needs to generate 16 bits per clock (8I, 8Q)
 - ◆ 32 Users at 125 MHz (The generator supports 64 users @125 MHz)
 - ◆ Easily scale to 64 users @ 250 MHz with single engine
 - ◆ 32-64 user expansion is only a function of memory
- Assumptions
 - ◆ The same Gold Code output may be shared by the different fingers of a single channel if the fingers are aligned
 - ◆ Each Gold Code is unique per user

UMTS Gold Code Generator as defined by 3G TS 25.213



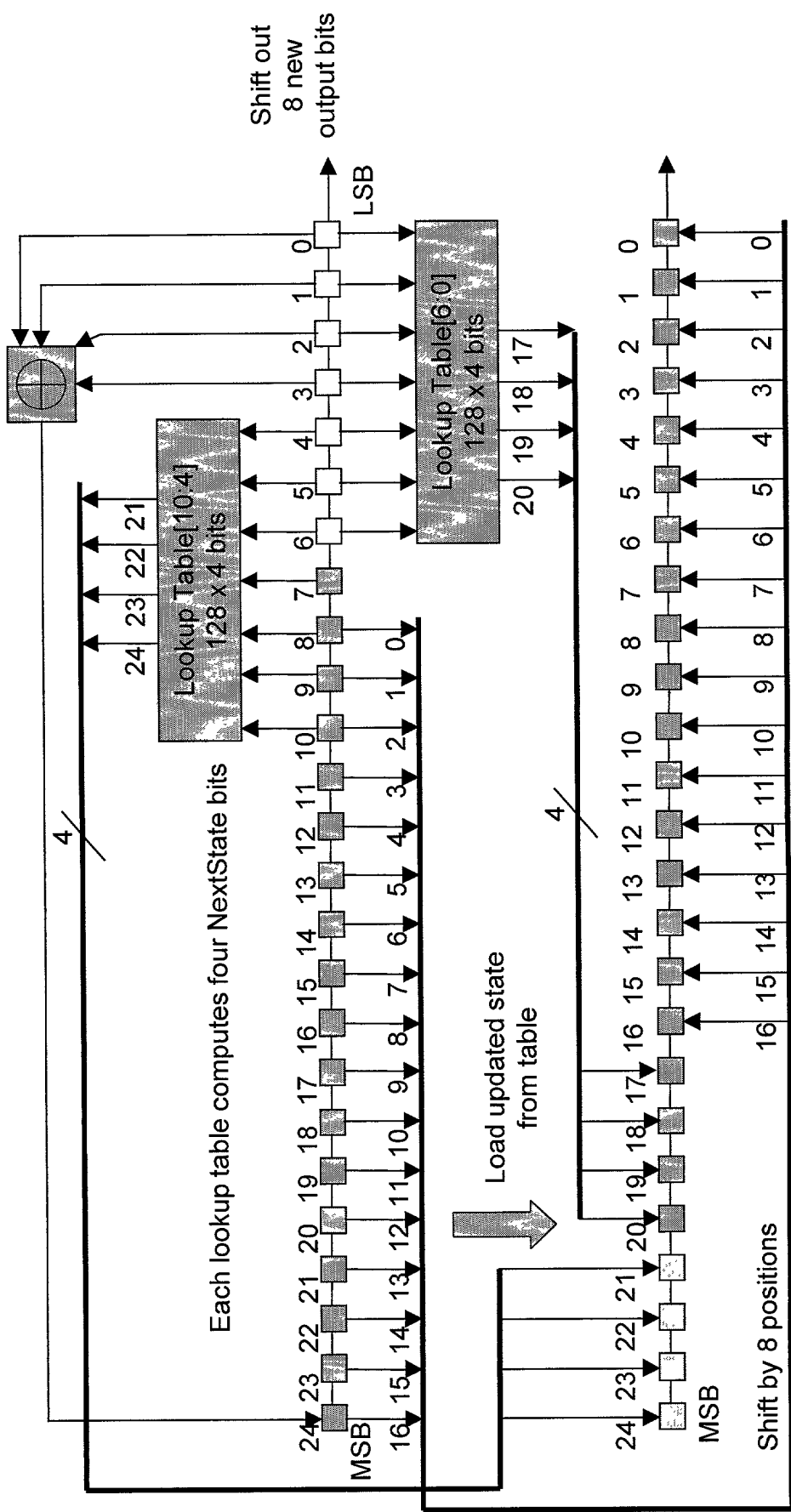
Exploit Code Properties to Eliminate Masks



Linear Feedback Shift Register (LFSR) Initial Values

- Each LFSR is reset to its initial value at the beginning of the frame
- The seed for LFSRA is assigned by the Network Controller
- The seed for LFSRB is 0x1FFFFFFF for all users
- The seed for LFSRC is the contents of LFSRA's seed shifted by 16,777,232 cycles, and is computed by the ARC at the beginning of the call
- The seed for LFSRD is 0x1FFFFFFF shifted by 16,777,232 cycles for all users and is static
- The seed values for all LFSRs are stored in LSMs

Lookup Table Determines the Next 8 Bits In a Single Cycle



8 chips per clock

Gold Code Generator Output Computations

$$C_{\text{long1},n} = \text{LSFRA}[7:0] \text{ XOR } \text{LSFRB}[7:0]$$

Let us define $\text{LFSRC}'[i] = \text{LSFRC}[2\lfloor i/2 \rfloor]$

$$C_{\text{long},n}(i) = C_{\text{long1},n}(i)(1 + j(-1)^i(C_{\text{long2},n}(2\lfloor i/2 \rfloor)) \quad (\text{from 3G TS25.213})$$

Multiplying bits by $+1/-1$ is the same as XOR for 0s and 1s.
XORing by 0xAA can be used in place of the $(-1)^i$ term.

In binary representation, the Scrambling Code $C_{\text{long},n}$ becomes:

$$C_{\text{long},n}[7:0] = C_{\text{long1},n}[7:0](1 + j(0xAA) \text{ XOR } C_{\text{long2},n}[7:0])$$

$$C_{\text{long},n}[7:0] = \text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0]$$

$$+ j(\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0] \text{ XOR } 0xAA \text{ XOR } \text{LFSRC}'[7:0] \text{ XOR } \text{LFSRD}'[7:0])$$

$$C_{\text{long},n}[7:0] = \text{SCI}[7:0] + j\text{SCQ}[7:0]$$

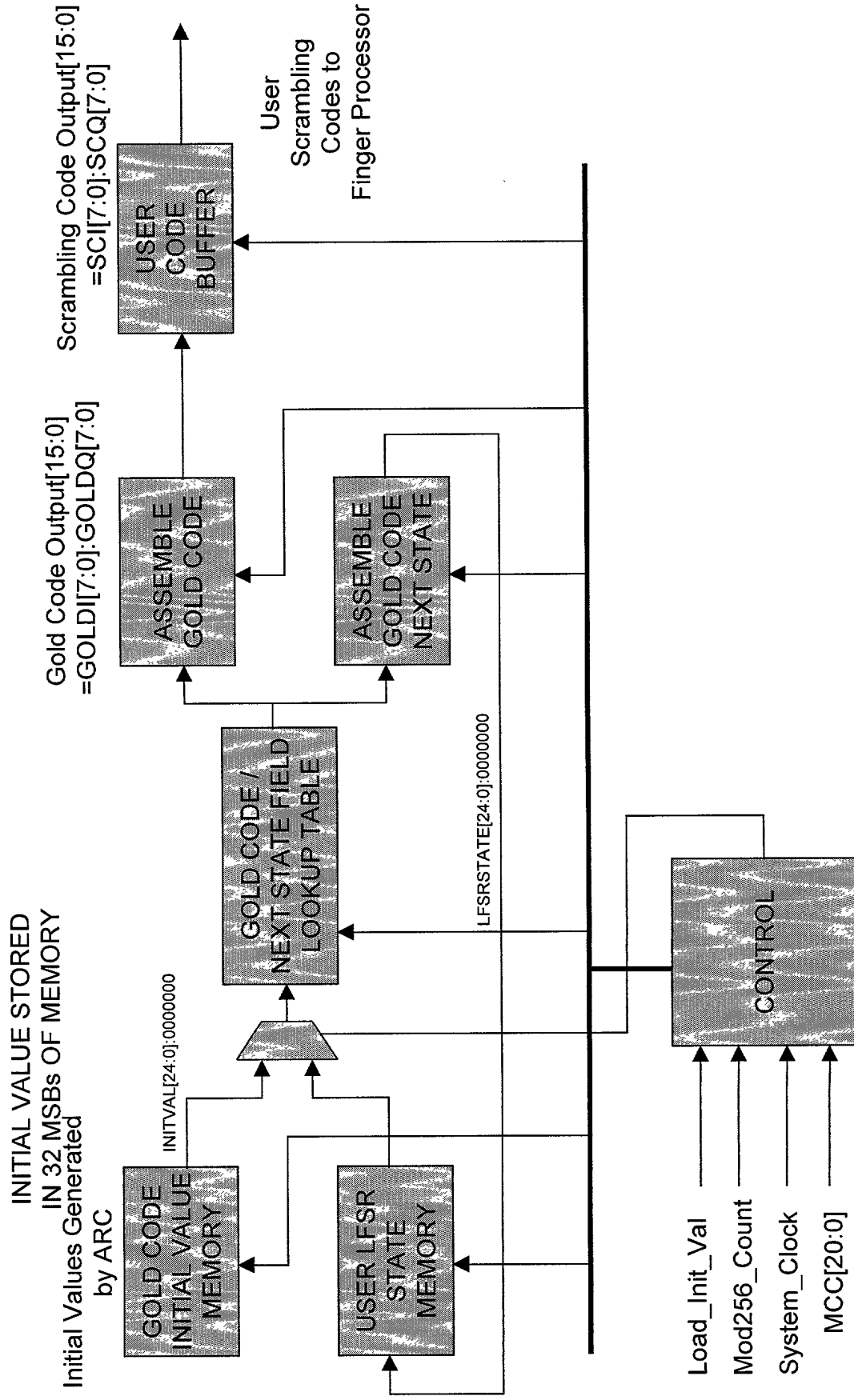
Let us define $\text{LFSRD}''[7:0] = 0xAA \text{ XOR } \text{LFSRD}'[7:0]$, then:

$$C_{\text{long},n}[7:0] = (\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0])$$

$$+ j(\text{LFSRA}[7:0] \text{ XOR } \text{LFSRB}[7:0] \text{ XOR } \text{LFSRC}'[7:0] \text{ XOR } \text{LFSRD}''[7:0])$$

We use a lookup table to compute $\text{LFSRC}'[7:0]$ and $\text{LFSRD}''[7:0]$

Gold Code Generator Functional Block Diagram



Gold Code Generator Memory Layout

Gold Code Initial Value Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	•
	•
	•
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User LFSR State Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	•
	•
	•
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User Code Buffer Memory

Address	Contents
0x1FE	User 63 PONG
0x1FC	User 62 PONG
0x1FA	User 61 PONG
	•
	•
	•
0x104	User 2 PONG
0x102	User 1 PONG
0x100	User 0 PONG
0xFE	User 63 PING
0x0C	User 62 PING
	•
	•
	•
0x04	User 2 PING
0x02	User 1 PING
0x00	User 0 PING

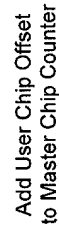
Gold Code Generator Lookup[6:0] Definitions

<p>At Address 4n+0: OUT[7:0] = Next StateA[3:0]:PASSA[3:0]</p> <p>OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]</p>	<p>At Address 4n+2: OUT[7:0] = Next StateC[3:0]:LFSRC[3:0]</p> <p>OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]</p>
<p>At Address 4n+1: OUT[7:0] = Next StateB[3:0]:PASSB[3:0]</p> <p>OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]</p>	<p>At Address 4n+3: OUT[7:0] = Next StateD[3:0]:LFSRD[3:0]</p> <p>OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0]</p>

Gold Code Generator Lookup[10:4] Definitions

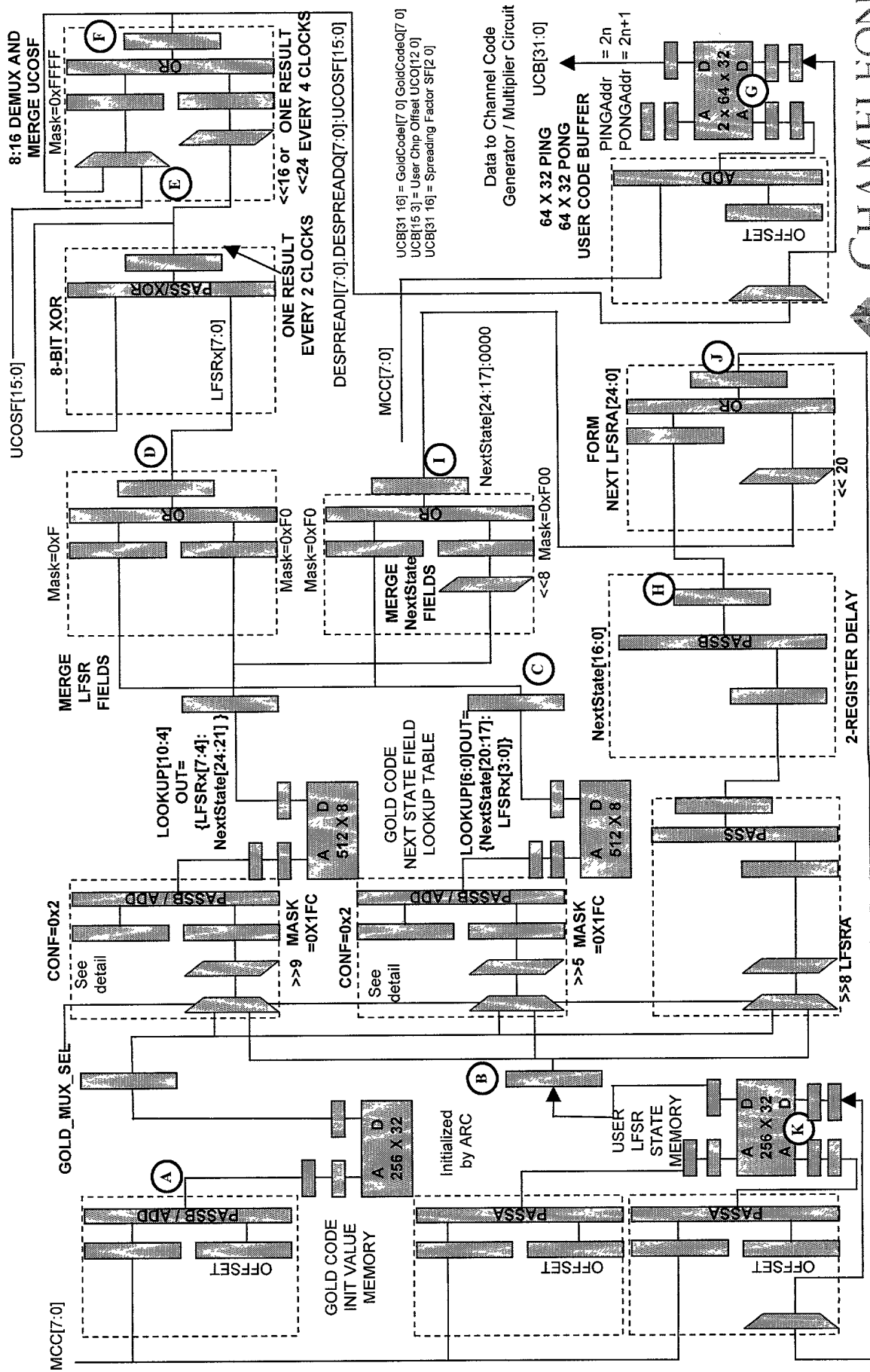
<p>At Address 4n+0: $OUT[7:0] = IN[7:4]:Next\ StateA[7:4]$</p> <p> $OUT[7] = IN[3]$ $OUT[6] = IN[2]$ $OUT[5] = IN[1]$ $OUT[4] = IN[0]$ $OUT[3] = IN[6] \text{ XOR } IN[3]$ $OUT[2] = IN[5] \text{ XOR } IN[2]$ $OUT[1] = IN[4] \text{ XOR } IN[1]$ $OUT[0] = IN[3] \text{ XOR } IN[0]$ </p>	<p>At Address 4n+2: $OUT[7:0] = IN'[7:4]:Next\ StateC[7:4]$</p> <p> $OUT[3] = IN[2]$ $OUT[2] = IN[2]$ $OUT[1] = IN[0]$ $OUT[0] = IN[0]$ $OUT[7] = IN[6] \text{ XOR } IN[3]$ $OUT[6] = IN[5] \text{ XOR } IN[2]$ $OUT[5] = IN[4] \text{ XOR } IN[1]$ $OUT[4] = IN[3] \text{ XOR } IN[0]$ </p>
<p>At Address 4n+1: $OUT[7:0] = IN[7:4]:Next\ StateB[7:4]$</p> <p> $OUT[7] = IN[3]$ $OUT[6] = IN[2]$ $OUT[5] = IN[1]$ $OUT[4] = IN[0]$ $OUT[3] = IN[6] \text{ XOR } IN[5] \text{ XOR } IN[4] \text{ XOR } IN[3]$ $OUT[2] = IN[5] \text{ XOR } IN[4] \text{ XOR } IN[3] \text{ XOR } IN[2]$ $OUT[1] = IN[4] \text{ XOR } IN[3] \text{ XOR } IN[2] \text{ XOR } IN[1]$ $OUT[0] = IN[3] \text{ XOR } IN[2] \text{ XOR } IN[1] \text{ XOR } IN[0]$ </p>	<p>At Address 4n+3: $OUT[7:0] = IN''[7:4]:Next\ StateD[7:4]$</p> <p> $OUT[3] = \neg IN[2]$ $OUT[2] = IN[2]$ $OUT[1] = \neg IN[0]$ $OUT[0] = IN[0]$ $OUT[7] = IN[6] \text{ XOR } IN[5] \text{ XOR } IN[4] \text{ XOR } IN[3]$ $OUT[6] = IN[5] \text{ XOR } IN[4] \text{ XOR } IN[3] \text{ XOR } IN[2]$ $OUT[5] = IN[4] \text{ XOR } IN[3] \text{ XOR } IN[2] \text{ XOR } IN[1]$ $OUT[4] = IN[3] \text{ XOR } IN[2] \text{ XOR } IN[1] \text{ XOR } IN[0]$ </p>

	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2
--	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	---

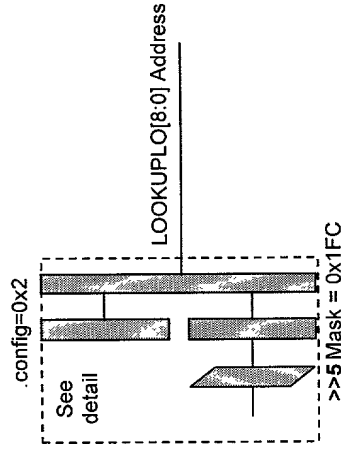


15

Gold Code Generator Implementation



Gold Code Generator Implementation Lookup[6:0] Address Generation DPU



Control needs to be able to select one of four lookup tables in the RAM.

Each lookup table is selected by two control inputs Select[1:0]:

Select[1:0] resides in the bottom two bits of the address

If Select[1:0]=0x0 then

ALU=PASSB

Address[1:0]=0x0

Output[7:4]= NextStateA[20:17], Output[3:0]=LFSRA[3:0]

If Select[1:0]=0x1 then

ALU=INC (B+1)

Address[1:0]=0x1

Output[7:4]= NextStateB[20:17], Output[3:0]=LFSRB[3:0]

If Select[1:0]=0x2 then

ALU=ADD (A + B)

Address[1:0]=0x2

Output[7:4]= NextStateC[20:17], Output[3:0]=LFSRC[3:0]

If Select[1:0]=0x3 then

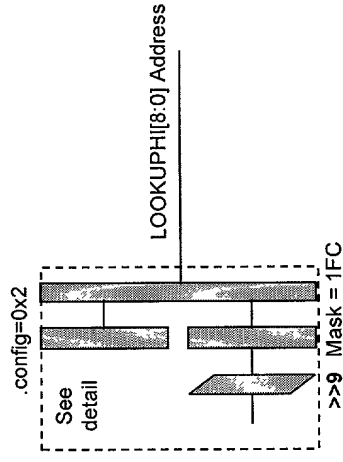
ALU=ADDCNT (A + B + 1, force Cin by control)

Address[1:0]=0x3

Output[7:4]= NextStateD[20:17], Output[3:0]=LFSRD[3:0]

Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

Gold Code Generator Implementation Lookup[10:4] Address Generation DPU



Control needs to be able to select one of four lookup tables in the RAM.

Each lookup table is selected by two control inputs Select[1:0]:

Select[1:0] resides in the bottom two bits of the address

If Select[1:0]=0x0 then

ALU=PASSB

Address[1:0]=0x0

Output[7:4]= LFSRA[7:4], Output[3:0]= NextStateA[24:21]

If Select[1:0]=0x1 then

ALU=INC (B+1)

Address[1:0]=0x1

Output[7:4]= LFSRB[7:4], Output[3:0]= NextStateB[24:21]

If Select[1:0]=0x2 then

ALU=ADD (A + B)

Address[1:0]=0x2

Output[7:4]= LFSRC[7:4], Output[3:0]= NextStateC[24:21]

If Select[1:0]=0x3 then

ALU=ADDCNT (A + B + 1, force C_n by control)

Address[1:0]=0x3

Output[7:4]= LFSRD[7:4], Output[3:0]= NextStateD[24:21]

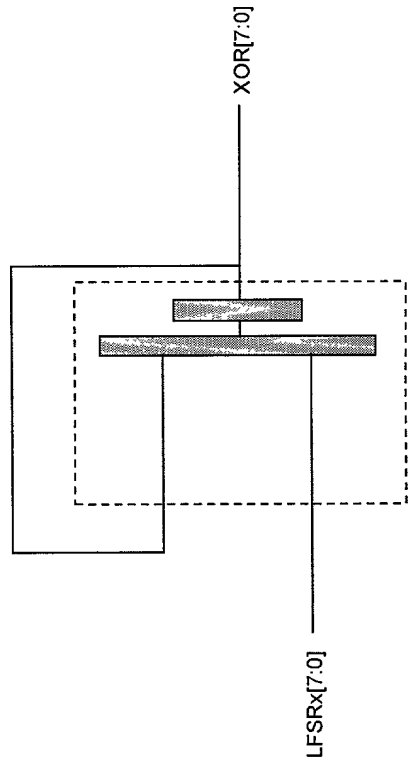
Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

Gold Code Generator Implementation

8-Bit XOR DPU

D	LFSR Input	A	B	C	D	A	B	C	D	A	B
	XOR DPU ALU Instruction	PASSB	XOR	XOR	XOR	PASSB	XOR	XOR	XOR	PASSB	XOR
E	XOR Output Register	$A \wedge B \wedge C \wedge D$	A	$A \wedge B$	$A \wedge B \wedge C$	$A \wedge B \wedge C \wedge D$	A	$A \wedge B$	$A \wedge B \wedge C$	$A \wedge B \wedge C \wedge D$	A

Where:
 $A = \text{LFSR}[7:0]$
 $B = \text{LFSR}[7:0]$
 $C = \text{LFSR}[7:0]$
 $D = \text{LFSR}[7:0]$
 $\wedge = \text{XOR}$



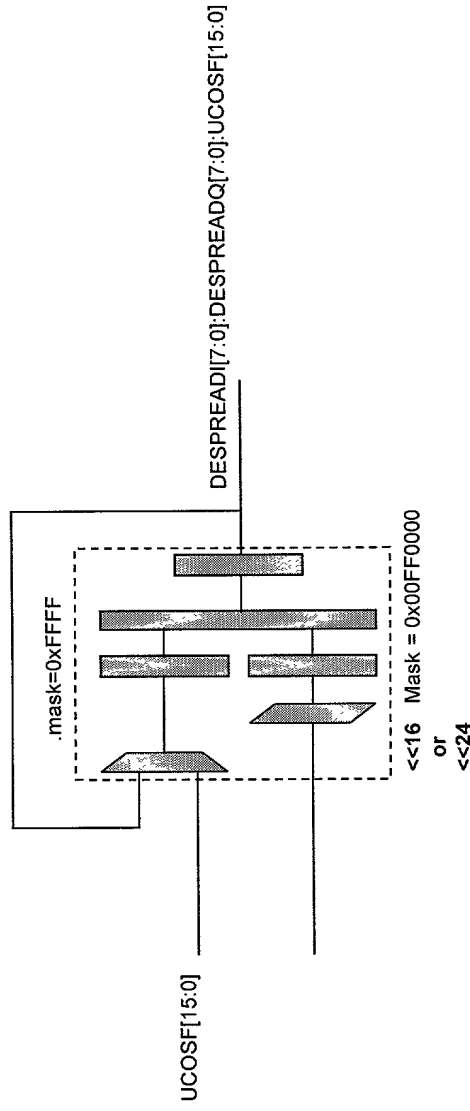
This DPU is used to perform an XOR operation

Control needs to generate two states for the DPU:

```

If PASS
    ;Pass the input to the output register
    ALUB=LFSRx[7:0] (from previous stage)
    ALU = PASSB
    OUTREGEN = 1
Else
    ;XOR the input with the output register
    ALUA = OUTREG
    ALUB=LFSRx[7:0] (from previous stage)
    ALU = XOR
    OUTREGEN = 1
    
```

Gold Code Generator Implementation 8:16 Demultiplex and Merge UCOSF DPU



This DPU is used to perform a 8:16 demultiplexing operation on the LFSR data as well as merge the UCOSF[15:0] field into the data stream before it is written into the User Code Buffer RAM.

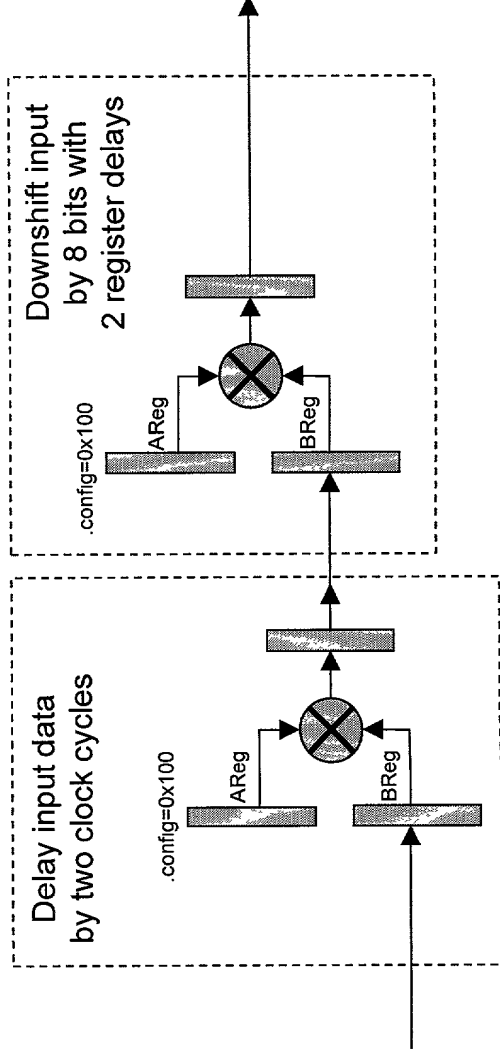
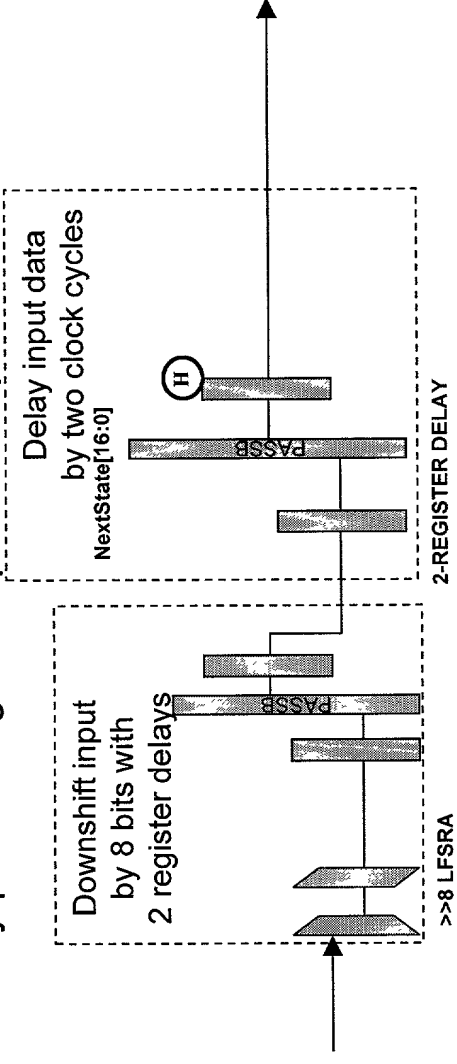
Control needs to generate three states for the DPU:

```

If MERGE_I
    ;Place UCOSF[15:0] in bits [15:0]
    ;and place DESPREADQ[7:0] in bits [31:24]
    ALUA=UCOSF[15:0] && (Mask==0xFFFF)
    ALUB=(DESPREADQ[7:0] from previous stage)<<24
    ALU = OR
    OUTREGEN = 1
If MERGE_Q
    ;Merge DESPREADQ[7:0] into bit positions [23:16]
    ALUA = OUTREG
    ALUB =(DESPREADQ[7:0] from previous stage)<<16 && Mask==0x00FF0000
    ALU = OR
    OUTREGEN = 1
Else
    OUTREGEN = 0
    
```

Gold Code Generator Implementation of Downshift Pipeline Delay with Multipliers

Two DPUs may be saved by performing the equivalent operations with two multipliers



Gold Code Generator Timing

A	Mod256_Count	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D	U4A	U4B	U4C	U4D
B	User LFSR State Output	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D
C	Gold Code Lookup Output	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
D	Merge LFSRx[7:0] Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
E	XOR LFSRx[7:0] Fields		U61Q		U62I		U62Q		U63I		U63Q		U0I		U0Q		U1I		U1Q		U2I
F	Merge Scrambling Codes			U61IQ				U62IQ				U63IQ					U0IQ				U1IQ
G	User to Finger Buffer Input	U60IQ				U61IQ			U62IQ					U63IQ				U0IQ			
H	Form NextState LFSRx[16:0]	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
I	Merge NextState Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
J	Form NextState LFSRx[24:0]	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B
K	User LFSR State Memory Input	U61A	U61B	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D

Gold Code Data Format:
Gold Code Output[15:0] = GOLDI[7:0]:GOLDQ[7:0]

UCSF[15:0] U0 U0 U62 U62 U1 U1 U63 U63 U2 U2 U0 U0 U3 U3 U1 U1 U4 U4 U2 U2

Gold Code Generator

UCOSF Output Timing

UCOSF[15:0]	U0	U0	U62	U62	U1	U1	U63	U63	U2	U2	U0	U0	U3	U3	U1	U1	U4	U4	U2	U2
UCC[12:0], UCC_EQUAL_0	XX	U0	U0	XX	XX	U1	U1	U63	XX	XX	U2	XX	XX	U3	U3	XX	XX	U4	U4	XX
REG_UCC_EQUAL_0	XX	XX	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U3	U3	U3	U3	U4	U4	U4
GOLD_MUX_SEL	U63	U63	U63	U0	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U2	U3	U3	U3	U4
Gold Code 8-BIT XOR	U61B		U62A		U62B		U63A		U63B		U0A		U0B		U1A		U1B		U2A	
UCOSF[15:0]			U62				U63				U0				U1				U2	
8:16 Demux Output		U61		U62 temp		U62		U63 temp		U63		U0 temp		U0	U1 temp		U1		U2 temp	

Gold Code Generator Resource Requirements

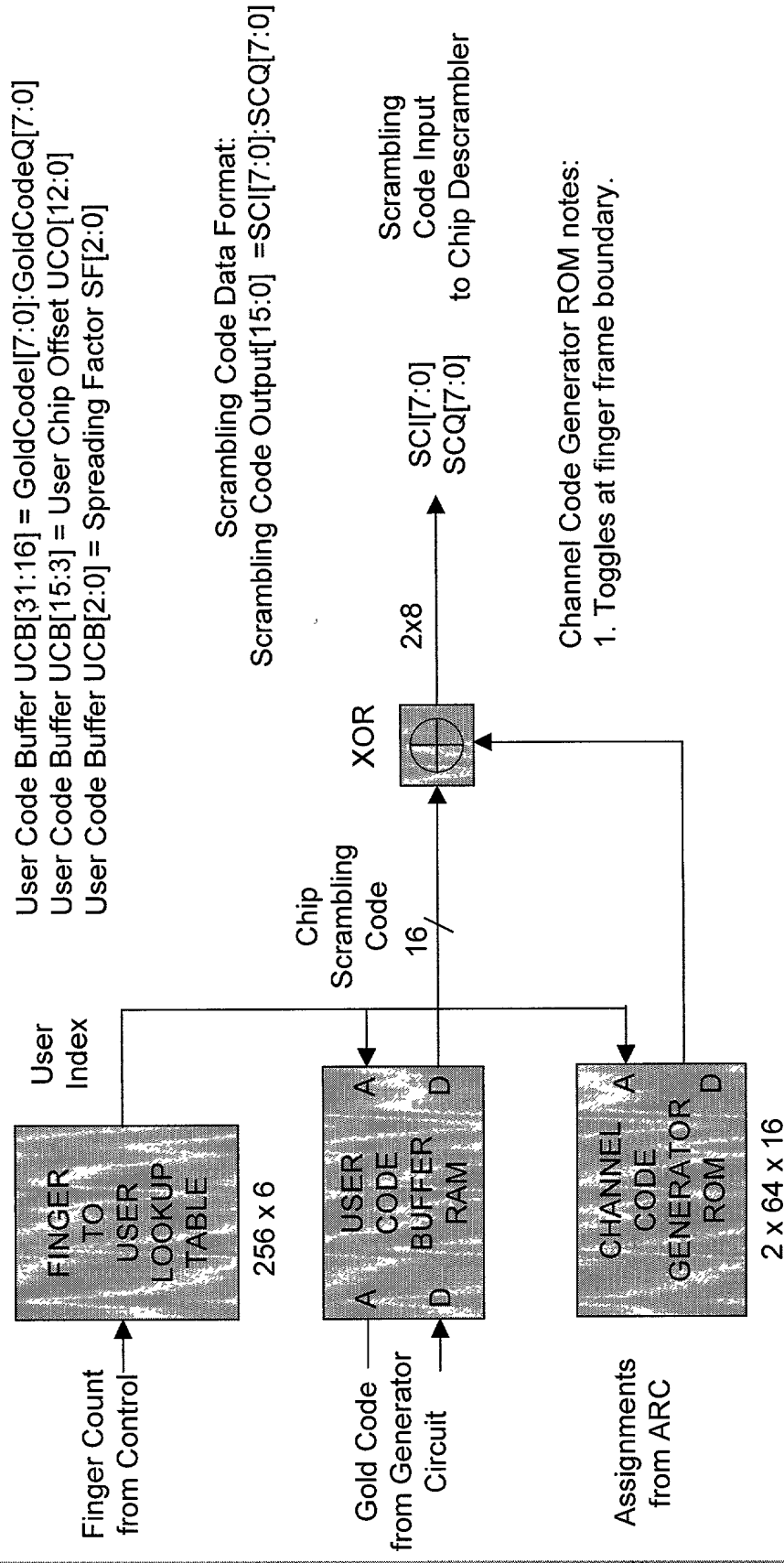
- 32 User Implementation @ 125 MHz
 - ◆ 14 DPUs
 - ◆ 8 LSMs
 - ◆ 2 Multipliers
- 64 User Implementation @ 125 MHz
 - ◆ 14 DPUs
 - ◆ 10 LSMs
 - ◆ 2 Multipliers
- 128 User Implementation @ 250 MHz
 - ◆ 16 DPUs
 - ◆ 12 LSMs
 - ◆ 2 Multipliers

Channel Code Generator / Multiplier Requirements and Assumptions

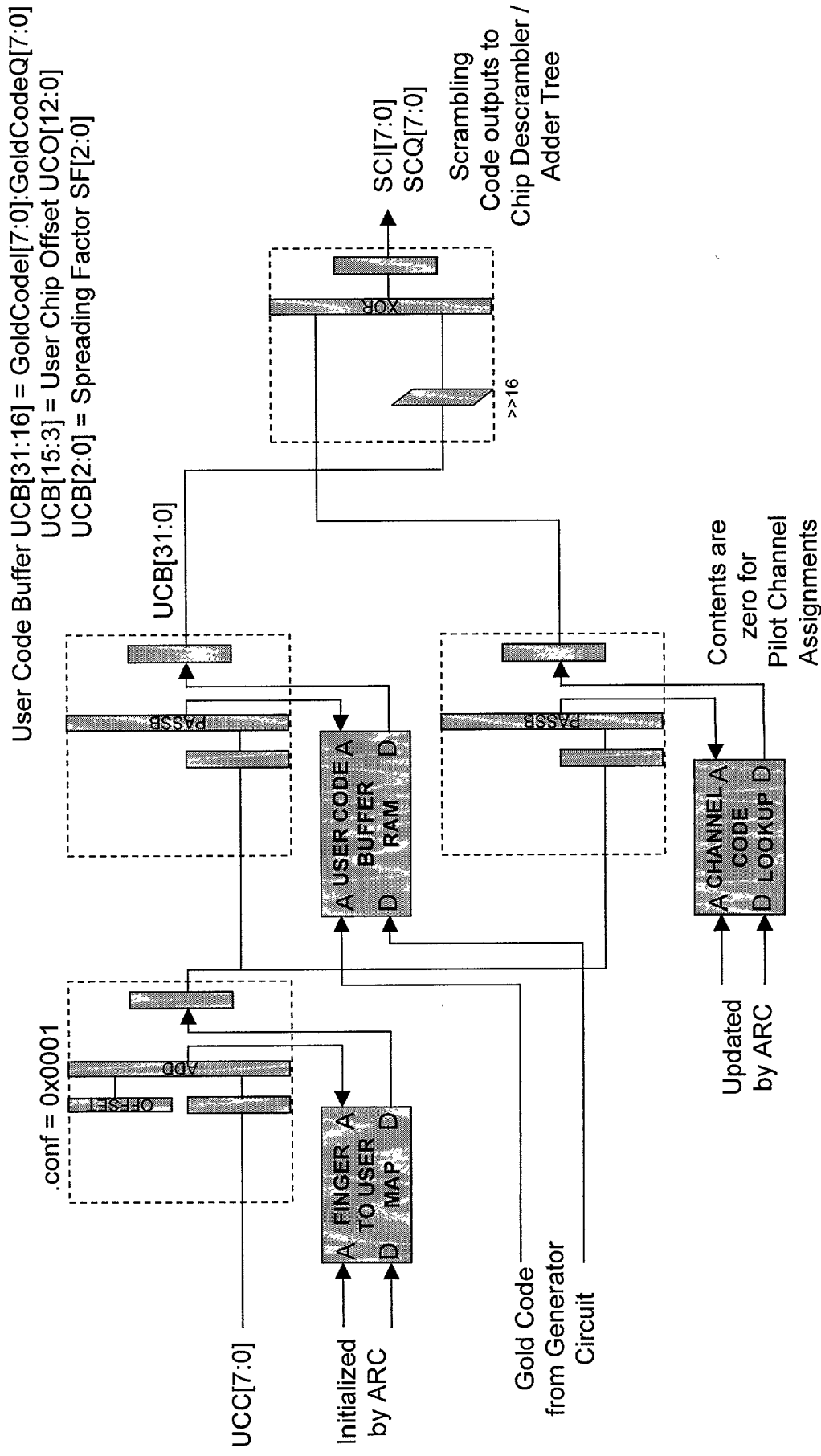
- Requirements
 - ◆ Provide a User to Finger Interface
 - ◆ Convert between User-based Gold Code and Finger-based Chip despreading
- Assumptions (from 3G TS25.213 Specification)
 - ◆ DPCCH $c_c = c_{ch,256,0} = 1, 1, 1, 1, \dots$ (all ones)
 - ◆ Single DPDCH $c_{d,1} = c_{ch,SF,k}$ where $k = SF/4$
 - ✦ $c_{ch,4,1} = 1, 1, -1, -1, \dots$
 - ✦ $c_{ch,8,2} = 1, 1, -1, -1, \dots$
 - ✦ $c_{ch,16,4} = 1, 1, -1, -1, \dots$
 - ◆ If more than one DPDCH $SF=4$, $c_{d,n} = c_{ch,4,k}$
 - ✦ $k=1$ if $n \in \{1,2\}$
 - ✦ $k=3$ if $n \in \{3,4\}$
 - ✦ $k=2$ if $n \in \{5,6\}$

Channel Code Generator / Multiplier

Functional Block Diagram



Channel Code Generator / Multiplier Implementation



Channel Code Generator / Multiplier Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 4 DPUs
 - ◆ 2 LSMs

- Note that the Gold Code Output Buffer LSM resources are counted in the Gold Code Generator circuit

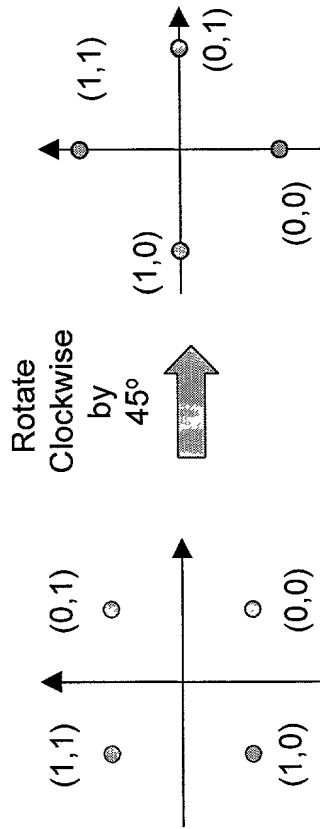
Chip Descrambler Requirements and Assumptions

- Requirements
 - ◆ Include the Adder Tree Input multiplexer
 - ◆ Descramble 8 chips per clock
 - ◆ Process 256 fingers @ 125 MHz
 - ◆ Process 512 fingers @ 250 MHz
- Assumptions
 - ◆ I must be able to read 8 consecutive samples (T_c apart) per clock from the Antenna Sample Buffer from any one antenna
 - ◆ An 8:1 Multiplexer is needed to align the input data to an even SF boundary point
 - ◆ All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks

Chip Descrambler Functional Description

- Read sixteen consecutive $T_c/2$ samples out of the Antenna Buffer corresponding to one finger, at the offset specified by the Path Searcher
- Mask out the unwanted half-chip samples
- Align the remaining eight samples (32 x 8 Barrel Shifter) with the Descrambling Code (Gold Code)
- Sign-extend the 8-bit data samples to 16 bits
- Multiply the eight aligned samples with the appropriate Despreading Codes

Chip Descrambler Functional Description



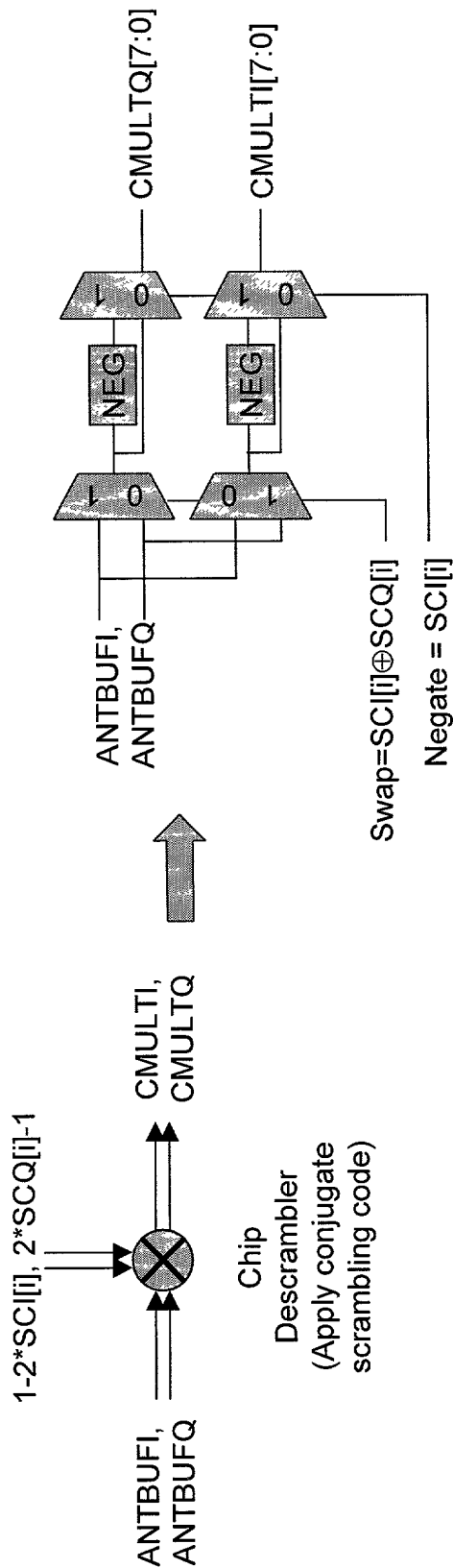
Labels:
(SCI[i], SCQ[i])

IN[31:16]=Q[15:0], IN[15:0]=I[15:0]

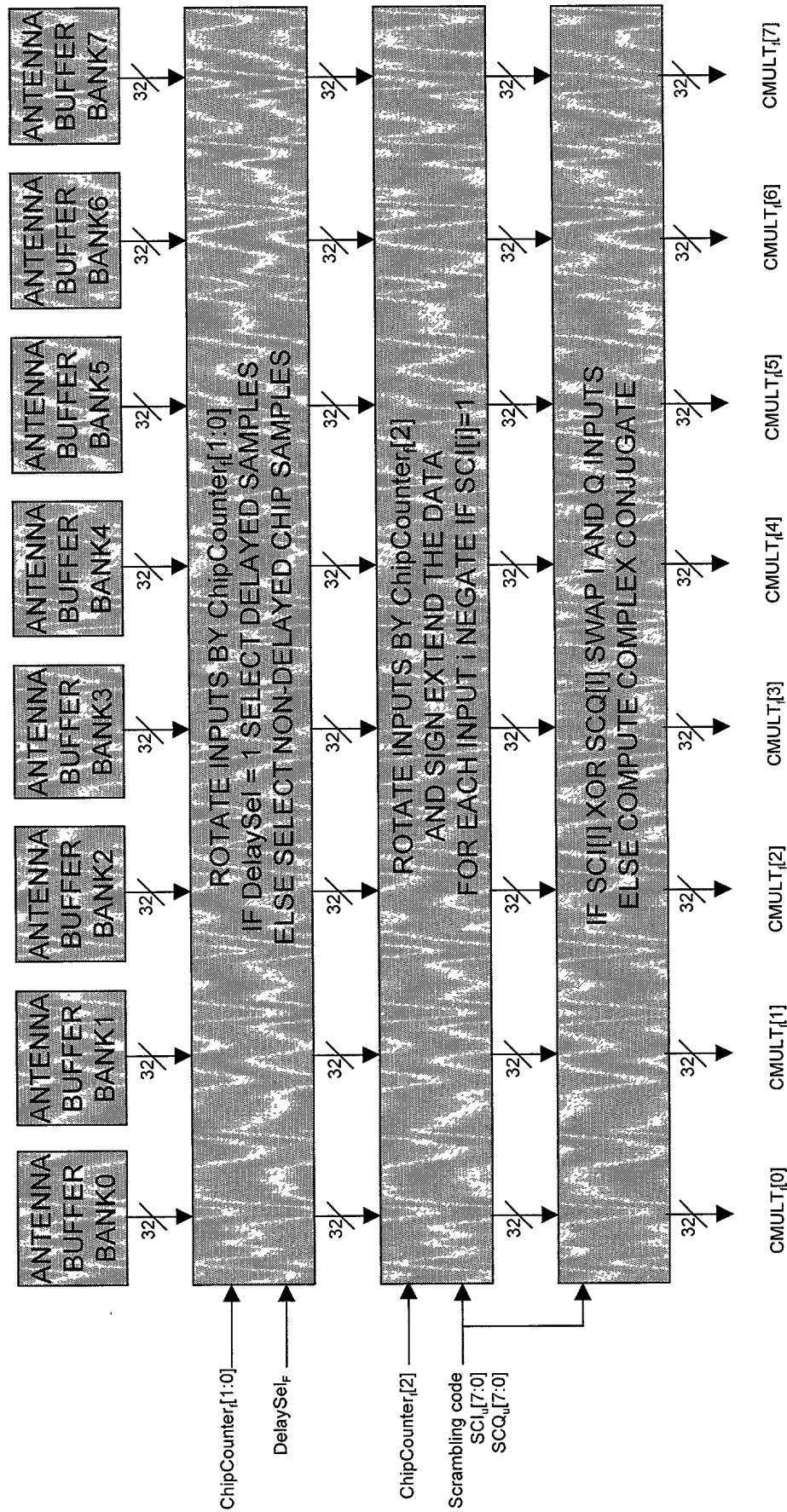
SCI	SCQ	OUT[31:16]	OUT[15:0]	SWAP/CONJUGATE	NEGATE
0	0	-I	-Q	Conjugate	
0	1	Q	I	Swap	
1	0	-Q	-I	Swap	Negate
1	1	I	Q	Conjugate	Negate

OUT[31:16]= CMULTI[15:0], OUT[15:0]=CMULTQ[15:0]

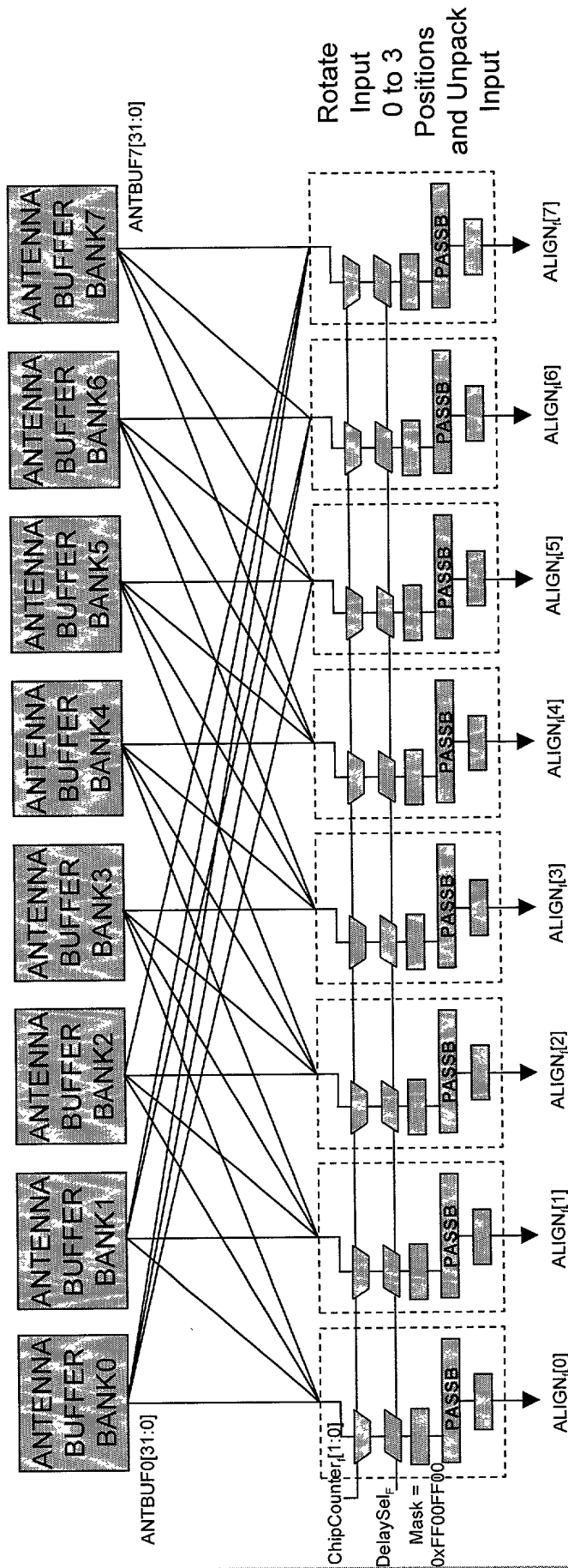
No multiplies or adds required



Chip Descrambler Block Diagram



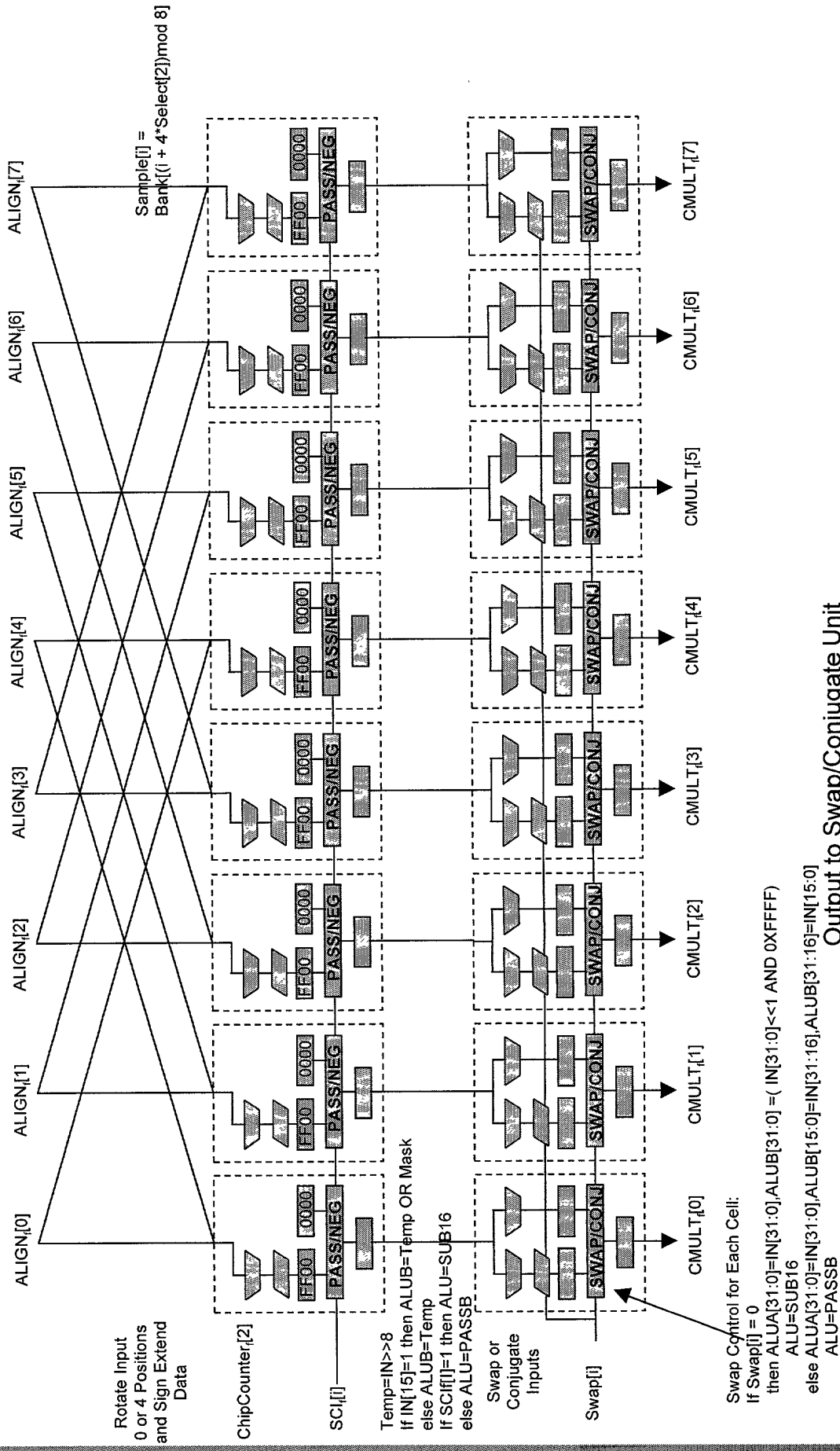
Chip Descrambler Input Alignment Implementation



Outputs to Conjugate Unit

Input Multiplexer Control:
 $ALIGN_i[j] = Bank[(i + Select[1:0]) \bmod 8]$

Chip Descrambler Chip Multiplier Implementation



Chip Descrambler Data Organization

Since the DPUs can only perform a SWAP or a Complex Conjugate, the I and Q inputs are assumed to be preswapped at the input to the Chip Descrambler circuit.

	SAMPLE[31:24]	SAMPLE[23:16]	SAMPLE[15:8]	SAMPLE[7:0]
SAMPLE BUFFER OUTPUTS: ANTBUF[31:0]	DQ _n [7:0]	DelayedDQ _n [7:0]	DI _n [7:0]	DelayedDI _n [7:0]
INPUT ALIGNMENT OUTPUTS: ALIGN _F [31:0]	DQ[7:0]	0x00	DI[7:0]	0x00
CHIP MULTIPLIER OUTPUTS: CMULT _F [31:0]	DI _n [15:0]	DI _n [7:0]	DQ _n [15:8]	DQ _n [7:0]
ADDER TREE OUTPUTS: ADD _F [31:0]	DI _n [15:0]	DI _n [7:0]	DQ _n [15:8]	DQ _n [7:0]

Where n is the sample n and n+ 1/2 is the next half-chip sample out of the Antenna Sample Buffer

Chip Descrambler Control Implementation

Control Input:
Scrambling Code Input $SCI_m[7:0]$, $SCQ_m[7:0]$
for User m

$SCI_m[i]$	$SCQ_m[i]$	$OUT_{T,m}[i]$	$OutQ_m[j]$
0	0	$DQ_k[7:0]$	$-Dl_k[7:0]$
0	1	$Dl_k[7:0]$	$DQ_k[7:0]$
1	0	$-Dl_k[7:0]$	$-DQ_k[7:0]$
1	1	$-DQ_k[7:0]$	$Dl_k[7:0]$

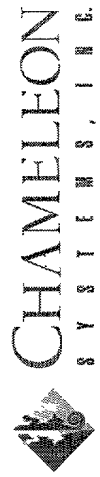
Data Inputs:
Eight Samples $DI_k[7:0]$
for $k = 0$ to 7

Chip Descrambler Resource Requirements

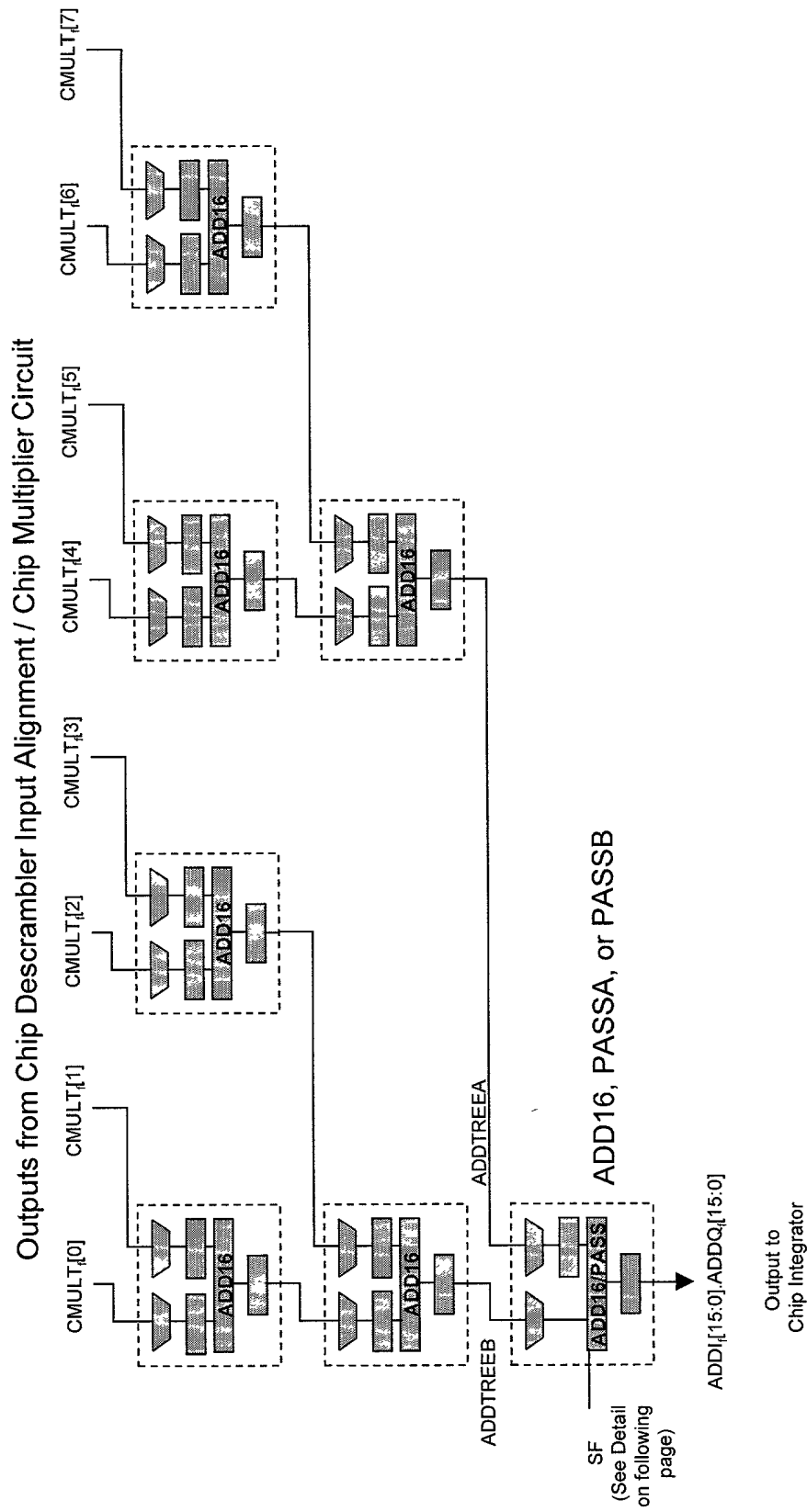
- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 24 DPUs
 - ◆ 0 LSMs

Chip Adder Tree Requirements and Assumptions

- Requirements
 - ◆ Add 8 chips per clock
 - ◆ Process 256 fingers @ 125 MHz
 - ◆ Process 512 fingers @ 250 MHz
- Assumptions
 - ◆ Inputs have been aligned on an even 8-chip boundary by the shifter in the Chip Descrambler circuit
 - ◆ All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks
 - ◆ Finger pairs allocated for SF=4 require that the first finger is assigned SF=4 and the second finger is assigned SF=0

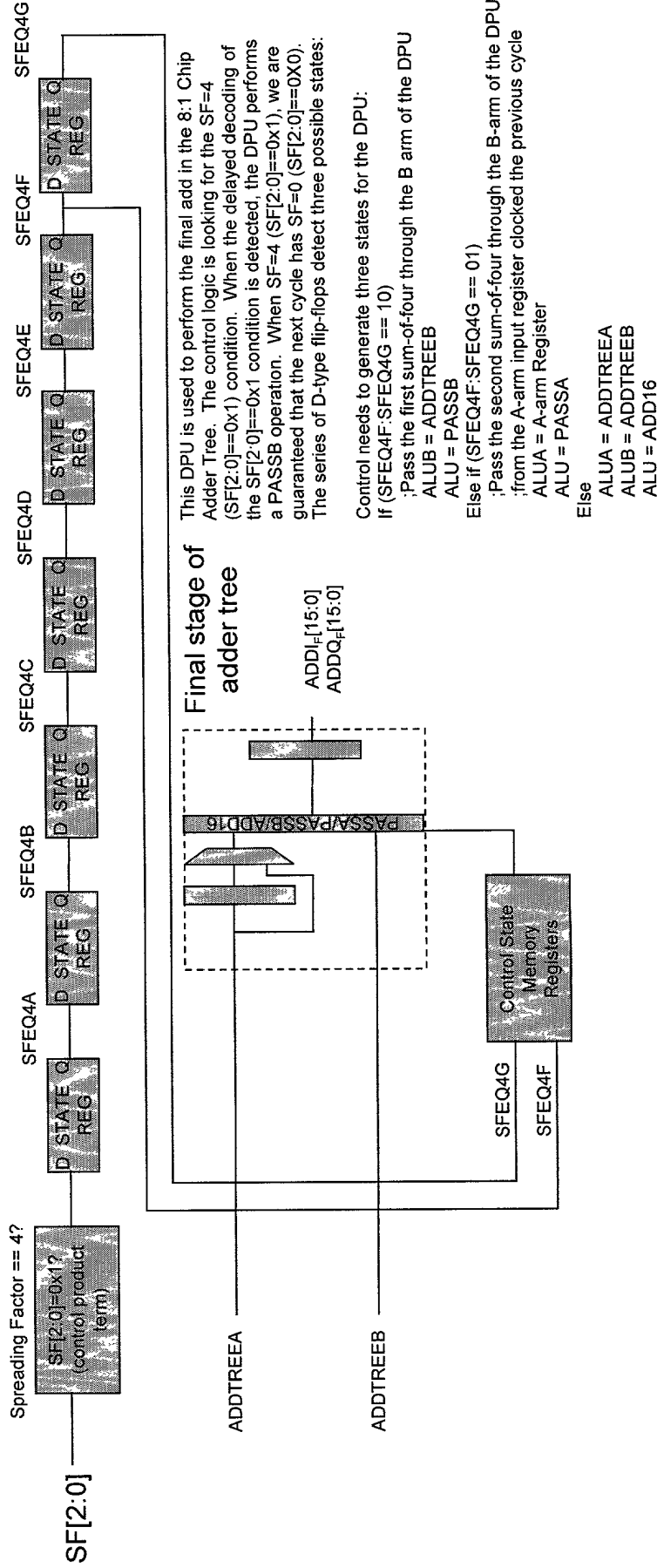
[illegible]

Chip Adder Tree Implementation



Chip Adder Tree Control Implementation

We need delayed versions of the Spreading Factor (SF) to control the Adder Tree



Chip Adder Tree Control Timing

F_n = Data for Finger n is valid

SF RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
UCB[2:0]=SF[2:0] valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4A=Reg. Decode of SF=4	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
SFEQ4B=Reg decode of SFEQ4A	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SFEQ4C=Reg decode of SFEQ4B	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
SFEQ4D=Reg decode of SFEQ4C	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
SFEQ4E=Reg decode of SFEQ4D	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
SFEQ4F=Reg decode of SFEQ4E	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage DPU CSR Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage ALU Control Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4G=Reg decode of SFEQ4F	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11

Chip Adder Tree Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 7 DPUs
 - ◆ 0 LSMs

Chip Integrator Requirements and Assumptions

- Requirements
 - ◆ Sum SF (Spreading Factor) partial sums into a single sum of SF chips
 - ◆ Prepare Data that is to be sent to the Channel Estimator in the ARC
- Assumptions
 - ◆ Average SF = 128
 - ◆ 256 chips input rate per 256 clocks @ 125 MHz

Chip Integrator

Theory of Operation

- Read one complex sum from Chip Adder Tree every clock cycle
- The LSB position of the Q component of the accumulated sum will be used to hold a valid bit for the backend circuits
- The LSB of the sum with $SF=4$ will have this LSB “robbed” without a noticeable effect on the result
- The accumulated sums with $SF=4-128$ will have their full-precision results shifted up one bit position
- For data with $SF=4-128$, shift input data up one bit position, for data with $SF=256$, pass data straight through
- Input data has already been despread by 8 chips (4 chips for fingers with $SF=4$)
- Sum input data with partial sum until SF chips ($SF/8$ inputs) have been added together

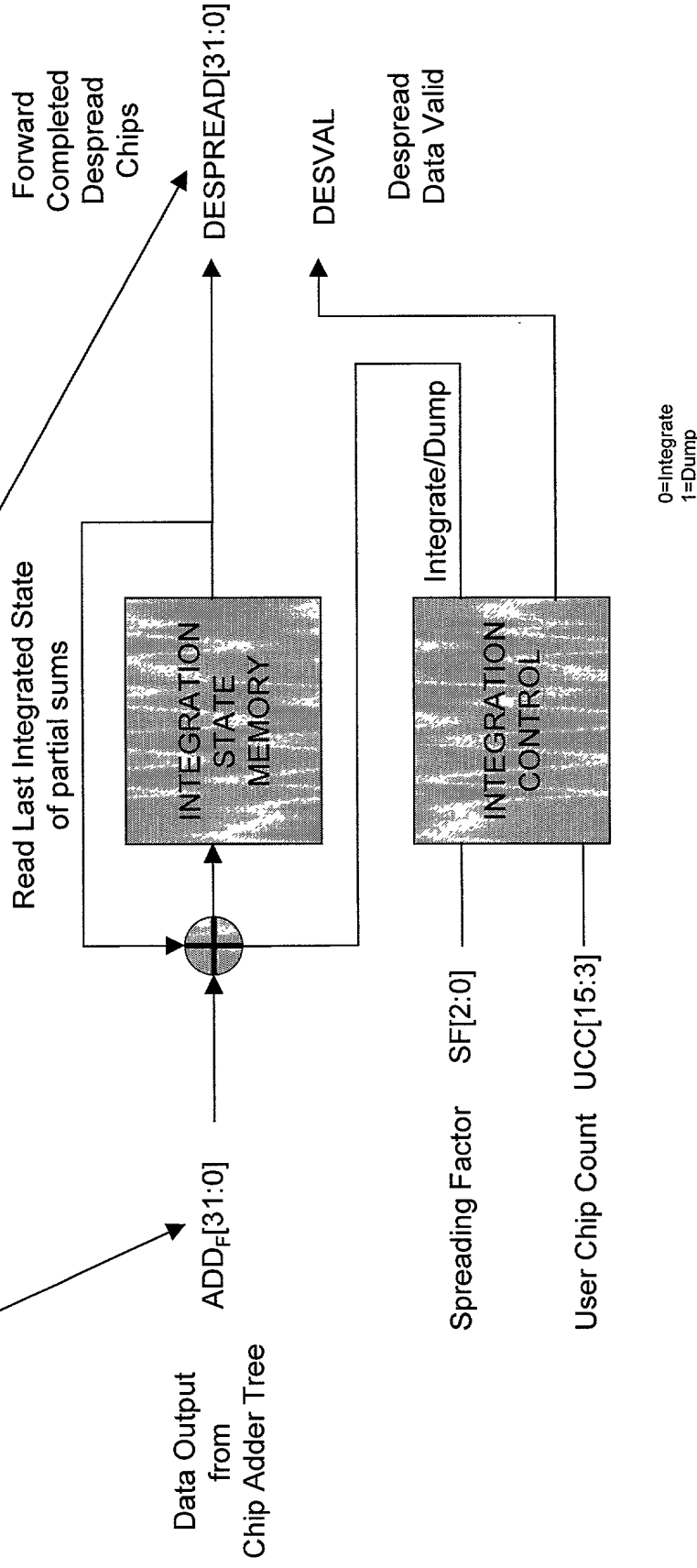
Chip Integrator Theory of Operation

- If the Chip Integrator input has $SF=4-128$ shift the input data up one bit position and set bits 0 and 16 to zero
- If the Chip Integrator input is the last 8-chip sample in a set of $SF/8$ chips, add it to the partial sum and place it into Chip Integrator Memory and set the valid bit (bit 0)
- If the Chip Integrator input is the first 8-chip sample in a set of $SF/8$ chips, place it into the Chip Integrator Memory and forward the previous despread sum

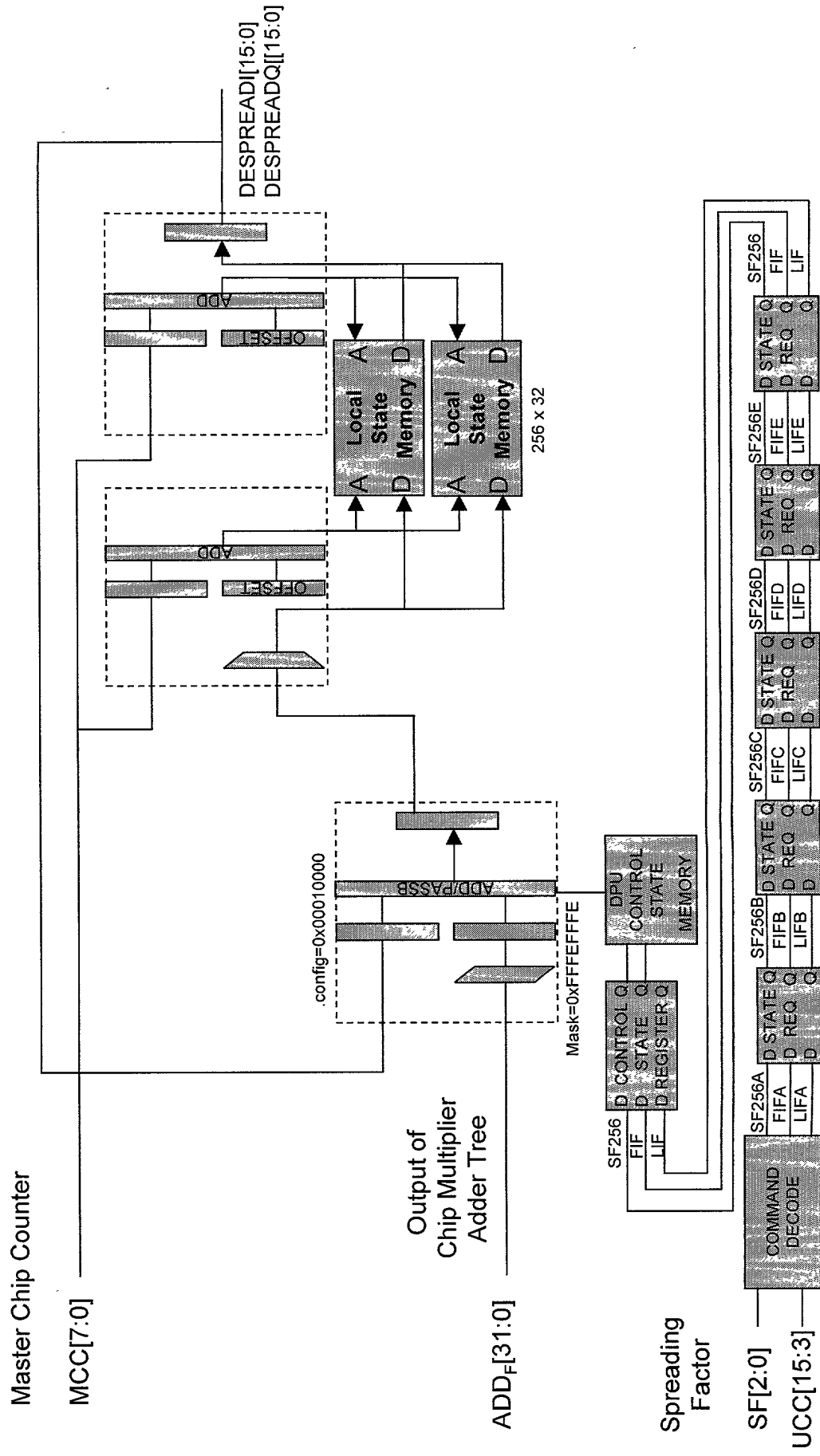
Chip Integrator Functional Block Diagram

$\text{DESPREAD}_F[31:0] = \text{DESPREADI}_F[14:0]:0:\text{DESPREADQ}_F[14:0]:\text{VALID}$

$\text{ADD}_F[31:0] = \text{ADDI}_F[15:0]:\text{ADDQ}_F[15:0]$



Chip Integrator Datapath Implementation



Pipeline delays to align the control with the data

Chameleon Systems Confidential

Chip Integrator Control Implementation (1/2)

First In Frame FIF = SF[2:0]==0 + SF[2:0]==1 + SF[2:0]==2 + (SF[2:0]==3 && UCC[3]==0x0) + (SF[2:0]==4 && UCC[4:3]==0x0) + (SF[2:0]==5 && UCC[5:3]==0x0) + (SF[2:0]==6 && UCC[6:3]==0x0) + (SF[2:0]==7 && UCC[7:3]==0x00)	;Spreading Factor = 0 ;Spreading Factor = 4 ;Spreading Factor = 8 ;Spreading Factor = 16 ;Spreading Factor = 32 ;Spreading Factor = 64 ;Spreading Factor = 128 ;Spreading Factor = 256
Last In Frame LIF = SF[2:0]==0 + SF[2:0]==1 + SF[2:0]==2 + (SF[2:0]==3 && UCC[3]==0x1) + (SF[2:0]==4 && UCC[4:3]==0x3) + (SF[2:0]==5 && UCC[5:3]==0x7) + (SF[2:0]==6 && UCC[6:3]==0xF) + (SF[2:0]==7 && UCC[7:3]==0x1F)	;Spreading Factor = 0 ;Spreading Factor = 4 ;Spreading Factor = 8 ;Spreading Factor = 16 ;Spreading Factor = 32 ;Spreading Factor = 64 ;Spreading Factor = 128 ;Spreading Factor = 256
Spreading Factor Equals 256 SF256 = SF[2:0]==7	
;Spreading Factor = 256	

Chip Integrator Control Implementation (1/2)

CASE (SF256:FIF:LIF)

- 000: ;Add 8-chip input to memory contents
 ALUA= AINPUT && Mask=0xFFFFEFFF
 ALUB= (BINPUT << 1) && Mask=0xFFFFEFFF
 ALU= ALUA + ALUB
- 001: ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit
 ALUA= AINPUT && Mask=0xFFFFEFFF
 ALUB= (BINPUT << 1) && Mask=0xFFFFEFFF
 ALU= ALUA + ALUB + 1 ;Set VALID bit
- 010: ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum
 ALUA= AINPUT && Mask=0xFFFFEFFF
 ALUB= (BINPUT << 1) && Mask=0xFFFFEFFF
 ALU= PASSB
- 011: ;Add 8-chip input to Chip Integrator Memory contents, set VALID bit,
 ;and forward previously despread sum
 ALUA= AINPUT && Mask=0xFFFFEFFF
 ALUB= (BINPUT << 1) && Mask=0xFFFFEFFF
 ALU= ALUA + ALUB + 1 ;Set VALID bit

Chip Integrator Control Implementation (2/2)

CASE (SF256:FIF:LIF) (CONTINUED)

100: ;Add 8-chip input to memory contents

ALUA= AINPUT

ALUB= BINPUT

ALU= ALUA + ALUB

101: ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit

ALUA= AINPUT && Mask=0xFFFFFFF

ALUB= BINPUT && Mask=0xFFFFFFF

ALU= ALUA + ALUB + 1 ;Set VALID bit

110: ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum

ALUA= AINPUT

ALUB= BINPUT

ALU= PASSB

111: ;THIS INPUT COMBINATION IS NOT POSSIBLE

;ARBITRARILY ASSIGN THE SAME COMMAND AS CASE 110

ALUA= AINPUT

ALUB= BINPUT

ALU= PASSB

Chip Integrator Control Timing

F_n = Data for Finger n is valid

SF/UCC RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
SF/UCC valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F254	U62A	U62B	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Outputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
INTA=Reg Decode of SF/UCC	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
INTB= Reg decode of INTA	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
INTC= Reg decode of INTB	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
INTD= Reg decode of INTC	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
INTE= Reg decode of INTD	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid CSR Inputs	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
DataValid ALU Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid Output Register Valid	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Integrate/Dump Output Reg Val	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Integrate/Dump CSR Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Integrate/Dump ALU Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12

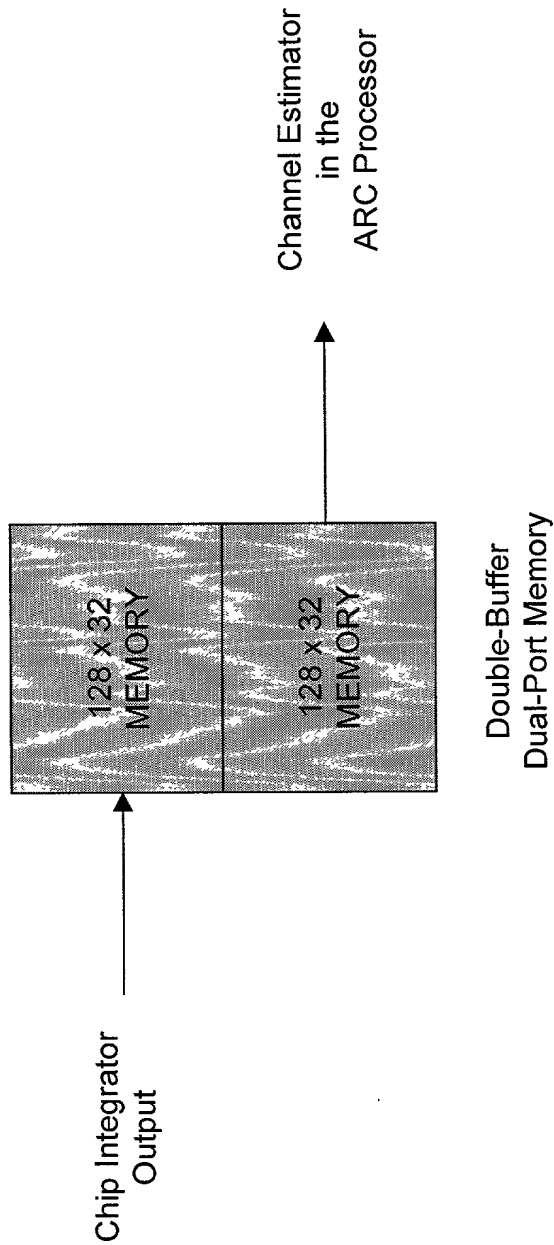
Chip Integrator Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 3 DPUs
 - ◆ 2 LSMs

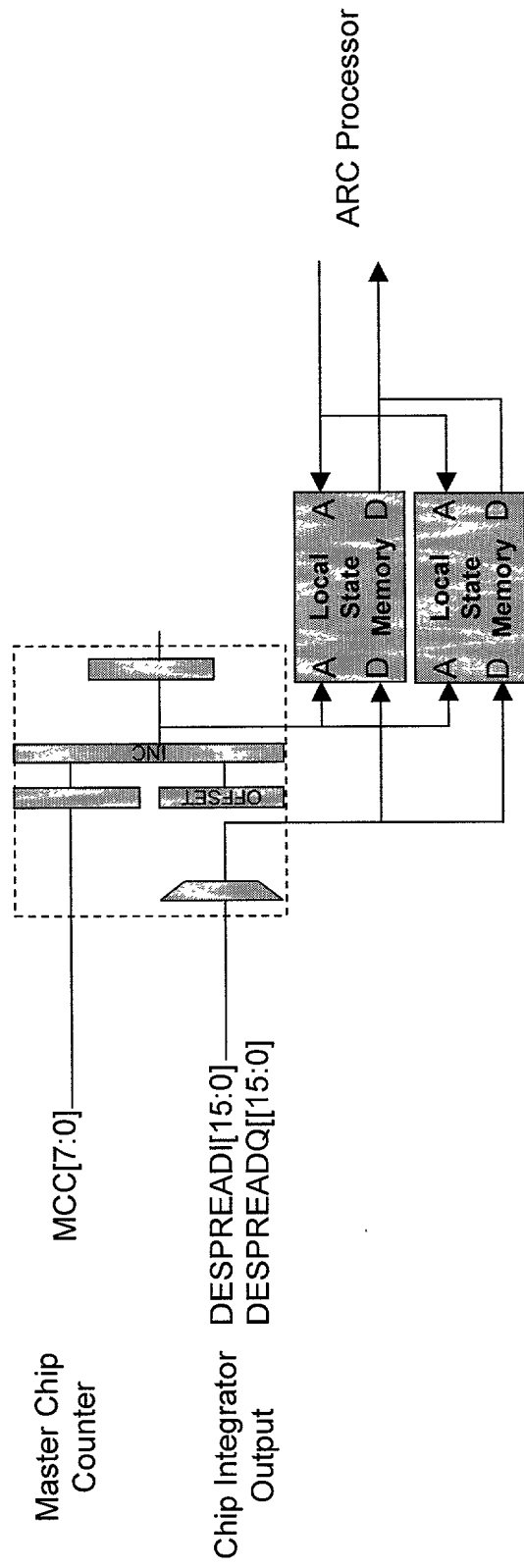
Channel Estimator Data Input Buffer Requirements and Assumptions

- Requirements
 - ◆ Provide a path between the Chip Integrator in the Chameleon fabric and the ARC processor
 - ◆ Double-buffer 128 despread pilot symbols every $256 T_c$
- Assumptions
 - ◆ Input written by Chip Accumulator
 - ◆ Output is read by the ARC core
 - ◆ All Control Channel data has SF=256

Channel Estimator Data Input Buffer Functional Block Diagram



Channel Estimator Data Input Buffer Implementation



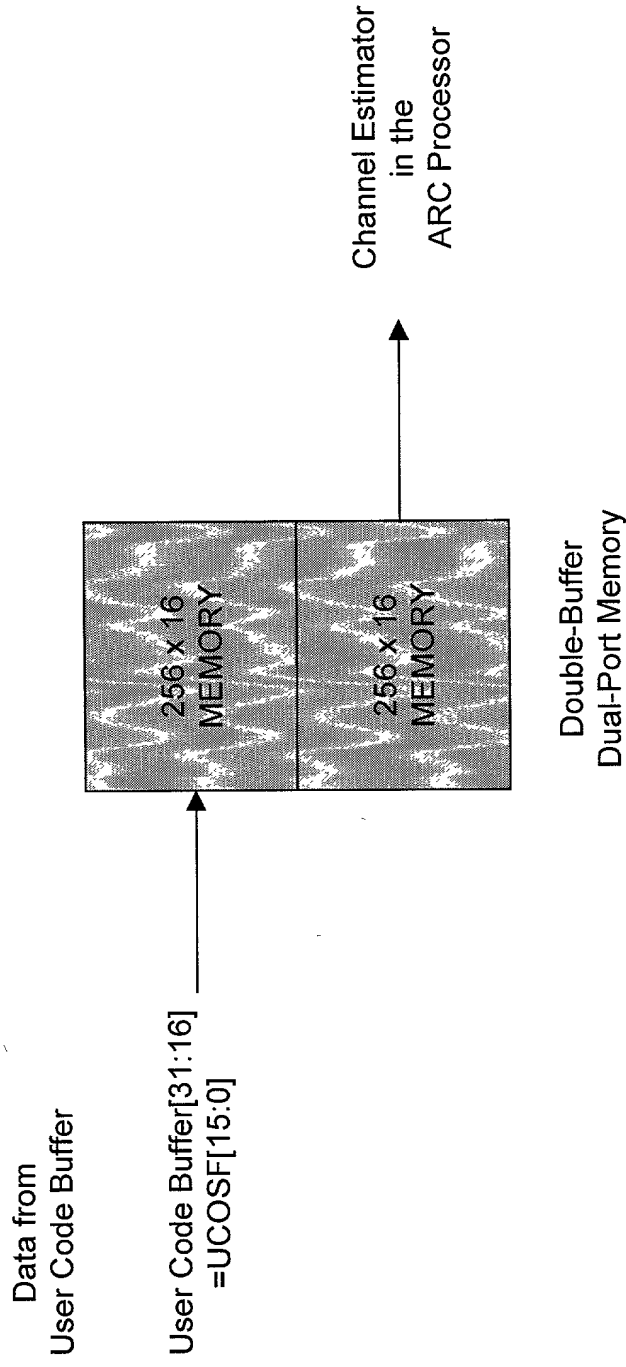
Channel Estimator Data Input Buffer Resource Requirements

- 32 User Implementation @ 125 MHz
 - ◆ 1 DPU
 - ◆ 2 LSMs
- 48 User Implementation @ 187.5 MHz
 - ◆ 1 DPUs
 - ◆ 4 LSMs
- 64 User Implementation @ 250 MHz
 - ◆ 1 DPUs
 - ◆ 6 LSMs

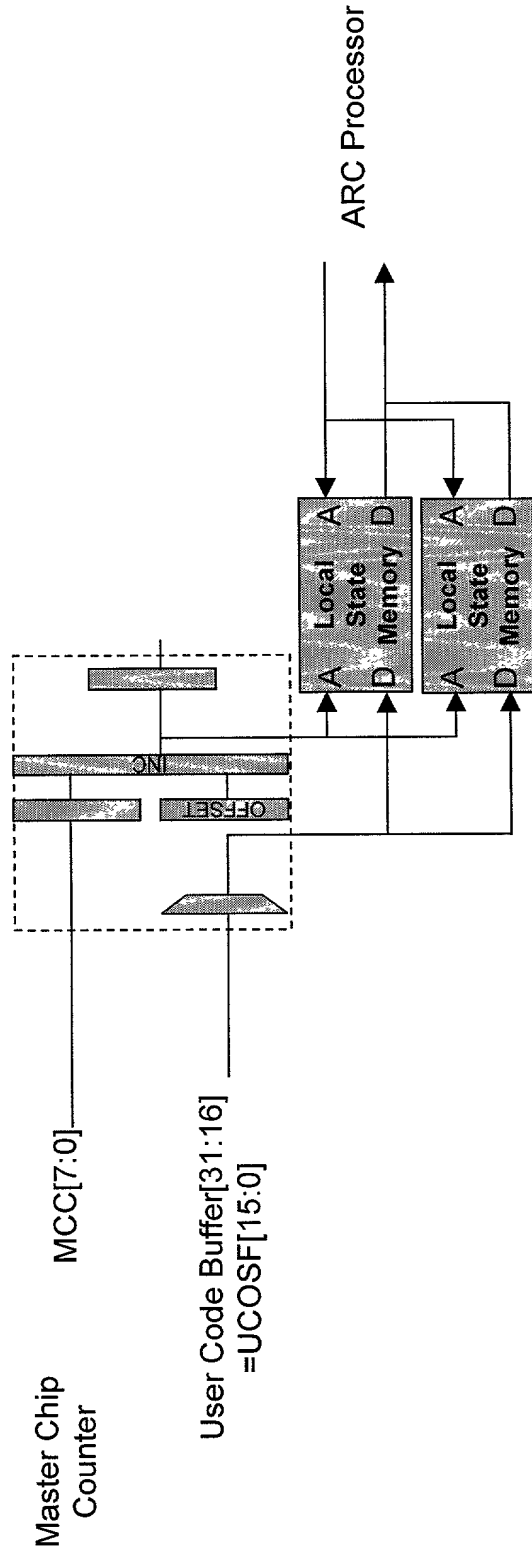
Channel Estimator UCC Input Buffer Requirements and Assumptions

- Requirements
 - ◆ Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Pilot chips
 - ◆ Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Data chips
 - ◆ Double-buffer 256 UCCs every $256 T_c$
- Assumptions
 - ◆ Input written by User Code Buffer
 - ◆ Output is read by the ARC core

Channel Estimator UCC Input Buffer Functional Block Diagram



Channel Estimator UCC Input Buffer Implementation



Channel Estimator UCC Input Buffer Resource Requirements

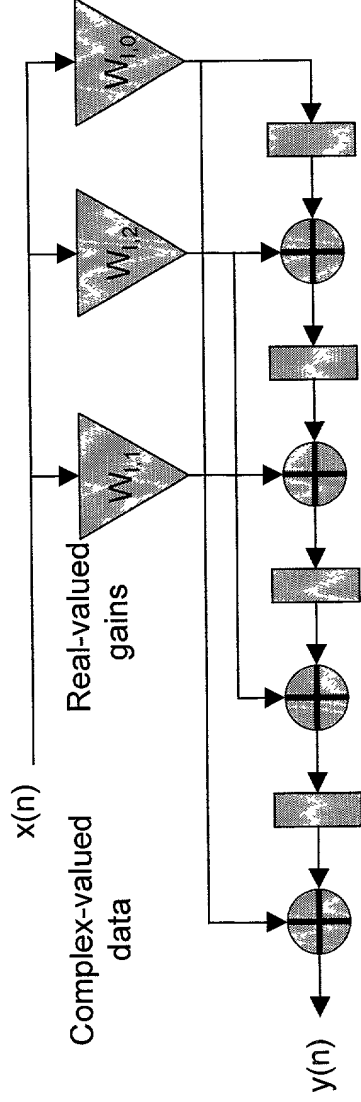
- 32 User Implementation @ 125 MHz
 - ◆ 1 DPU
 - ◆ 2 LSMs
- 48 User Implementation @ 187.5 MHz
 - ◆ 1 DPU
 - ◆ 4 LSMs
- 64 User Implementation @ 250 MHz
 - ◆ 1 DPU
 - ◆ 6 LSMs

Channel Estimator Requirements and Assumptions

- Requirements
 - ◆ Compensate the user data given the characteristics of the Pilot Channel data using XXX filtering
- Assumptions
 - ◆ All channel estimation is performed in the ARC processor
 - ◆ The Channel estimation is performed after the Pilot Channel has been despread (SF=256) and is significantly slower than the chip rate

FIR Filters for Channel Estimation

5-Tap Symmetric FIR Filter



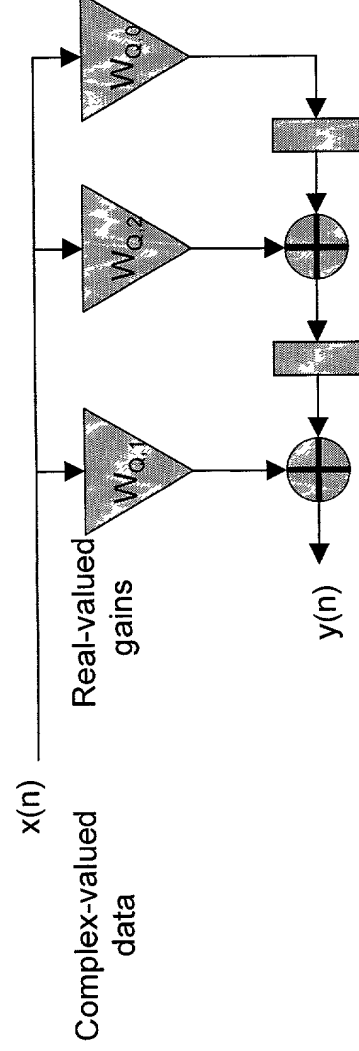
Required Ops/Sample:

- 6 Multiplies
- 4 Additions
- 3 Post-Multiply Packing Ops

Allow Multiple Clocks per Sample:

- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

3-Tap Non-symmetric FIR Filter



Required Ops/Sample:

- 6 Multiplies
- 2 Additions
- 3 Post-Multiply Packing Ops

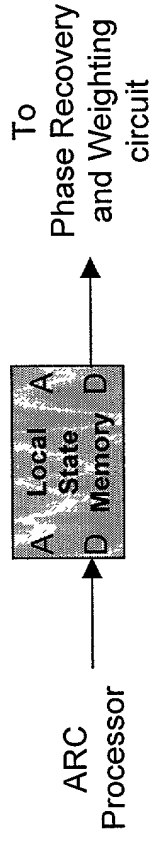
Allow Multiple Clocks per Sample:

- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

Channel Estimator Output Buffer Requirements and Assumptions

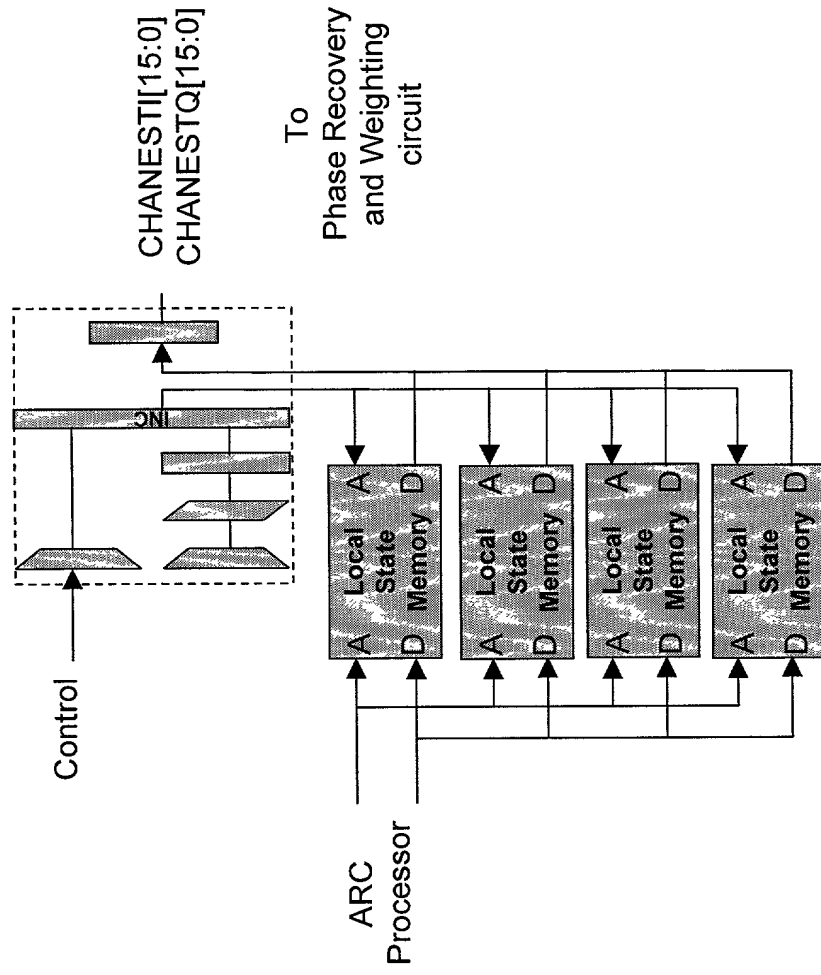
- Requirements
 - ◆ Provide a buffer between the Channel Estimation Filter and the Phase Recovery circuits
 - ◆ Provide a double buffer memory to prevent race conditions
- Assumptions
 - ◆ One 32-bit (16-bit I, 16-bit Q) Channel Compensation Weight word per pilot is required
 - ◆ 32 Users required
 - ◆ Must provide double buffer for proper operation
 - ◆ Channel Compensation word is sample and held on one time-slot ($2560 T_c$) boundaries

Channel Estimator Output Buffer Functional Block Diagram



256 Fingers: 256 fingers x 32-bit words x 2 banks (ping, pong)

Channel Estimator Output Buffer Implementation



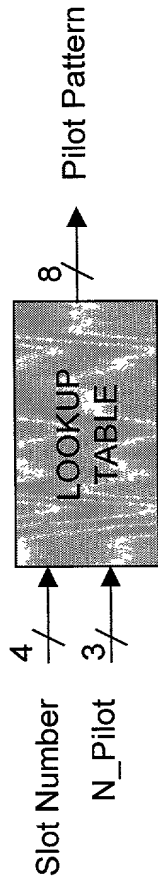
Channel Estimator Output Buffer Resource Requirements

- 32 Users Implementation
 - ◆ 1 DPU
 - ◆ 2 LSMs
- 48 Users Implementation
 - ◆ 2 DPU
 - ◆ 3 LSMs
- 64 Users Implementation
 - ◆ 2 DPU
 - ◆ 4 LSMs

Pilot Pattern Generator Requirements and Assumptions

- Requirements
 - ◆ Provide simple lookup-table approach
- Assumptions
 - ◆ Physically resides in ARC processor
 - ◆ N_Pilot = 3, 4, 5, 6, 7, or 8
 - ◆ The pilot is a function of slot
 - ◆ A single LSM is sufficient to contain the lookup-table

Pilot Pattern Generator Functional Block Diagram



Pilot Pattern Generator Memory Map Address Bit Definitions

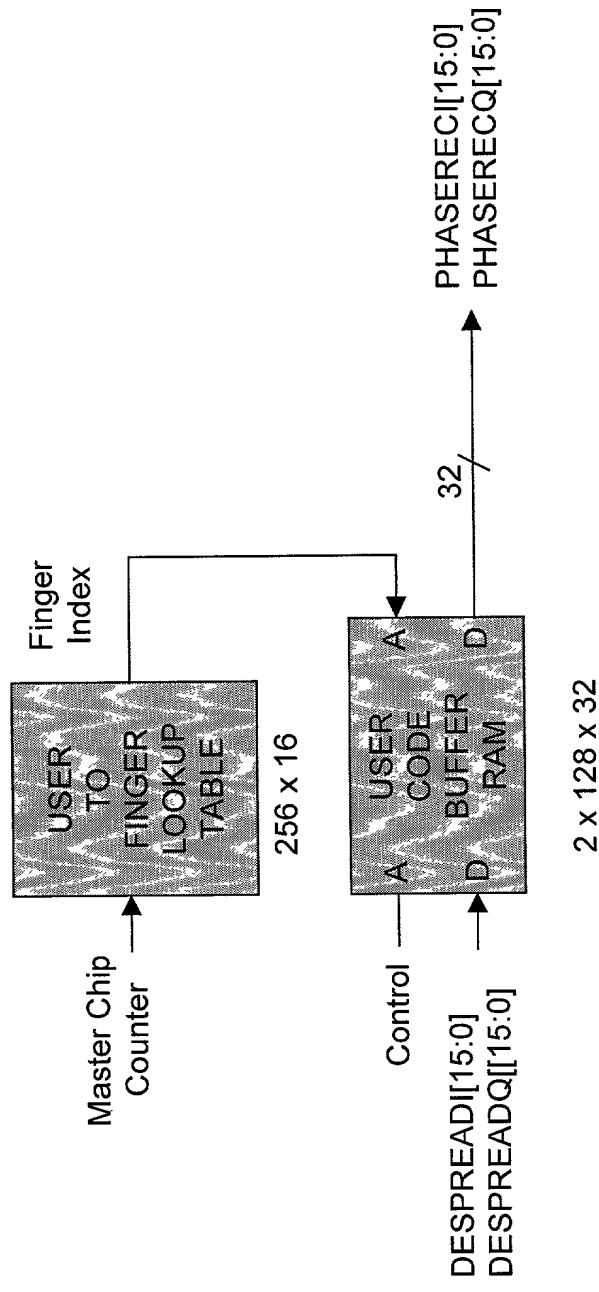
N Pilot	PilotGenAddr[6:4]
3	0
4	1
5	2
6	3
7	4
8	5

SlotNumber	PilotGenAddr[3:0]
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14

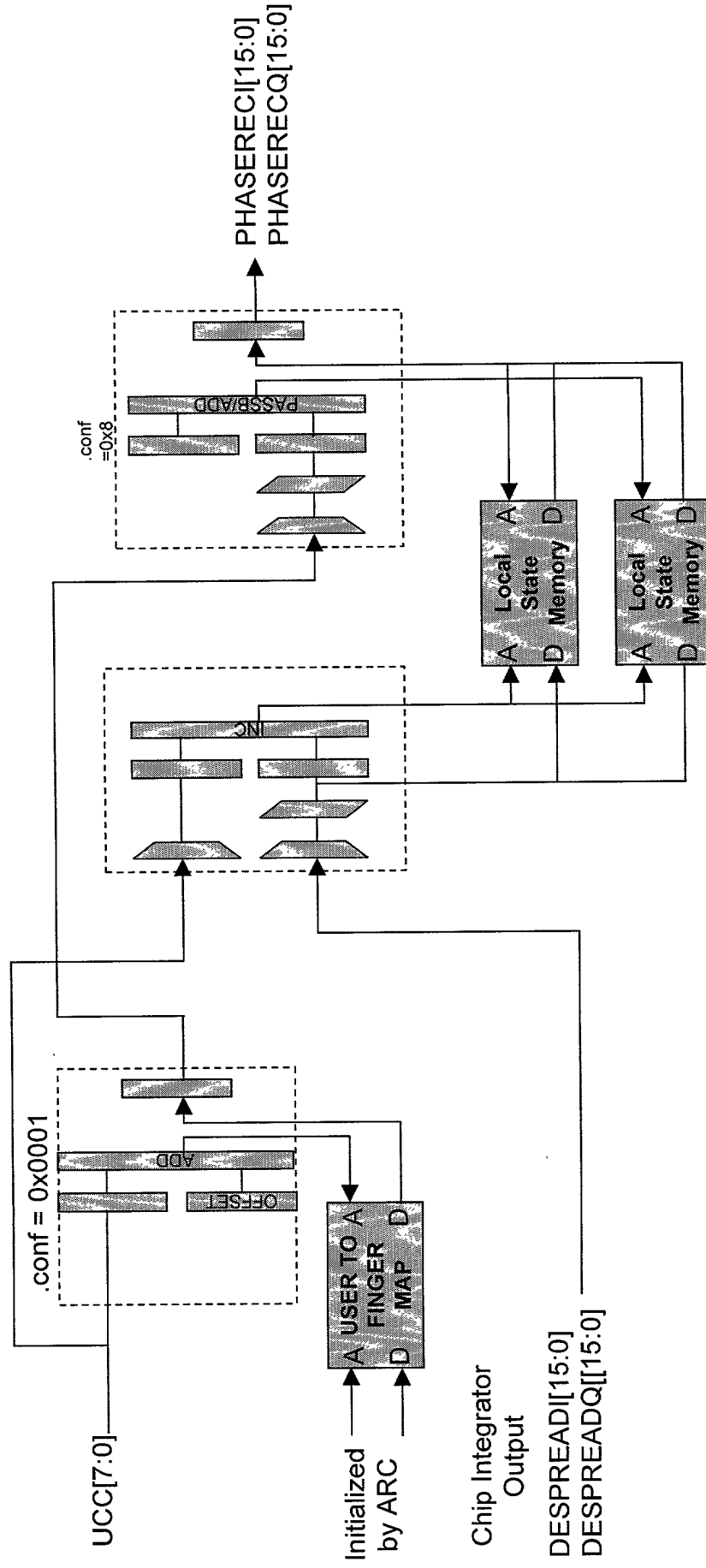
Phase Recovery Input Buffer Requirements and Assumptions

- Requirements
 - ◆ Provide a double buffer between the Chip Integrator circuit and the Phase Recovery circuit
 - ◆ Buffer 128 fingers every 256 clocks
- Assumptions
 - ◆ Only the delayed data channels are buffered
 - ◆ Pilot Channel data is not buffered

Phase Recovery Input Buffer Block Diagram



Phase Recovery Input Buffer Implementation



Phase Recovery and Weighting Requirements and Assumptions

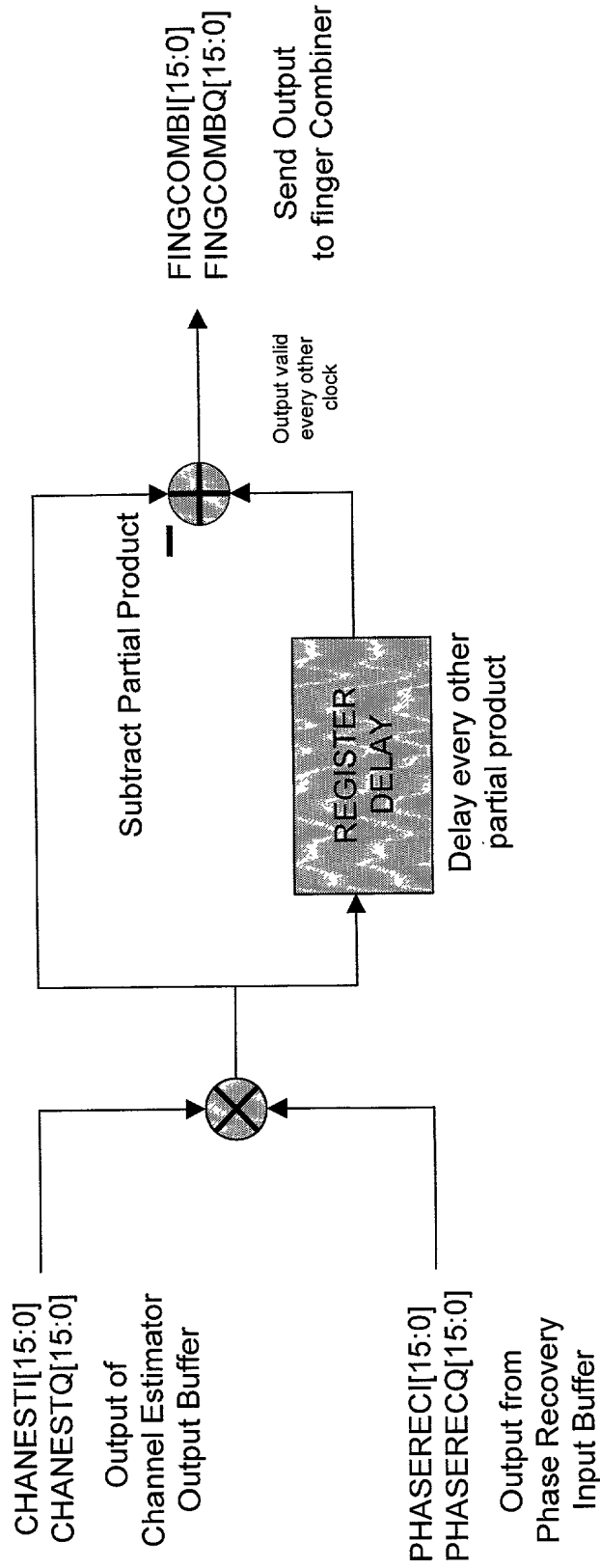
- Requirements
- Assumptions
 - ◆ A complex multiplication of $(A + jB) * (C + jD)$

$$= (AC - BD) + j(AD + BC)$$
 - ◆ We are only interested in the real part of the output:

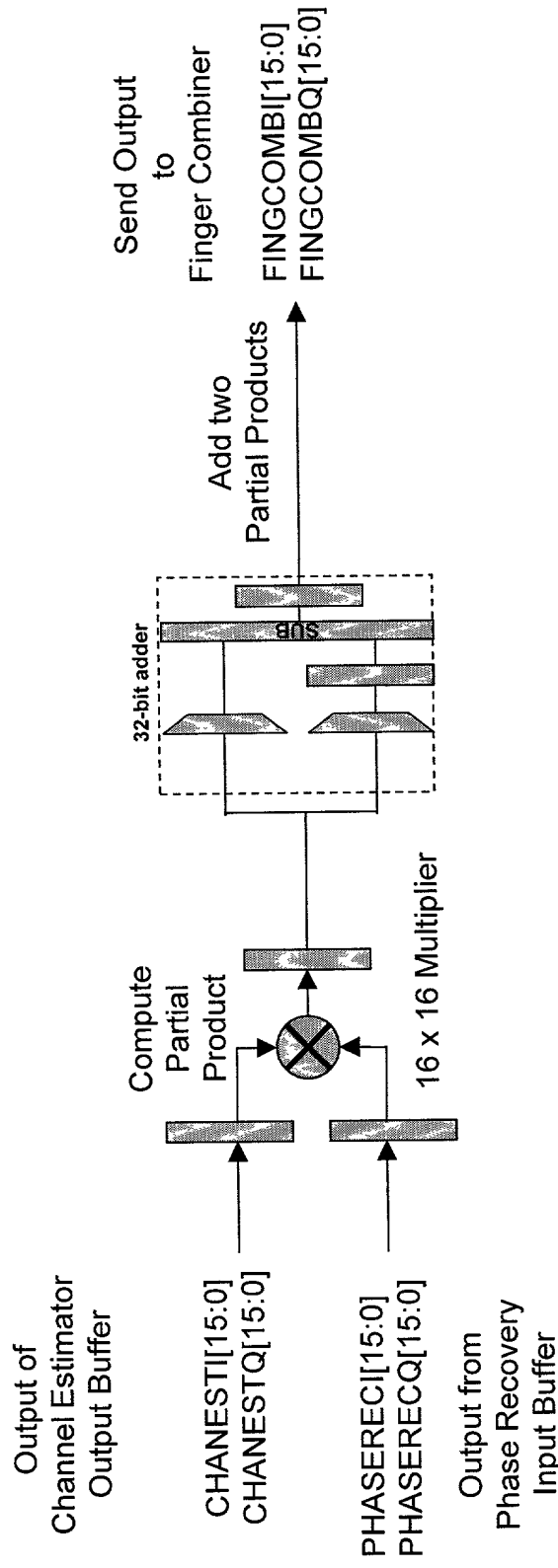
$$\text{Re}\{(A + jB) * (C + jD)\} = AC - BD$$

We have extra cycles so only one multiplier is required

Phase Recovery and Weighting Functional Block Diagram



Phase Recovery and Weighting Implementation



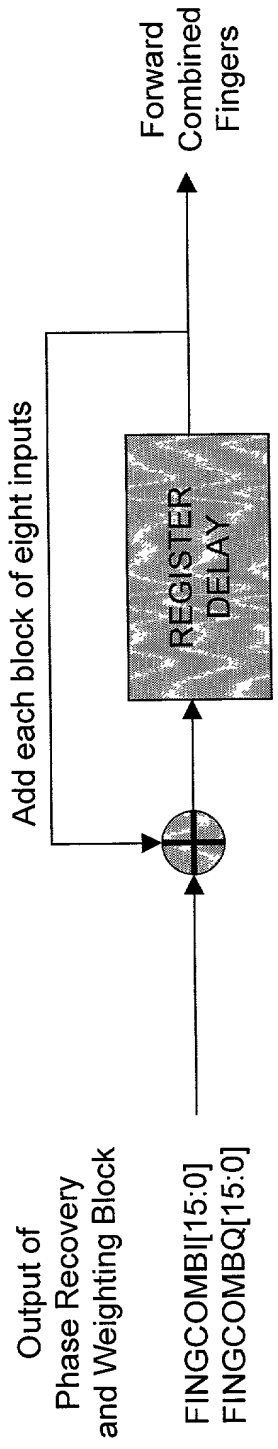
Phase Recovery and Weighting Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
 - ◆ 1 DPU
 - ◆ 1 Multiplier

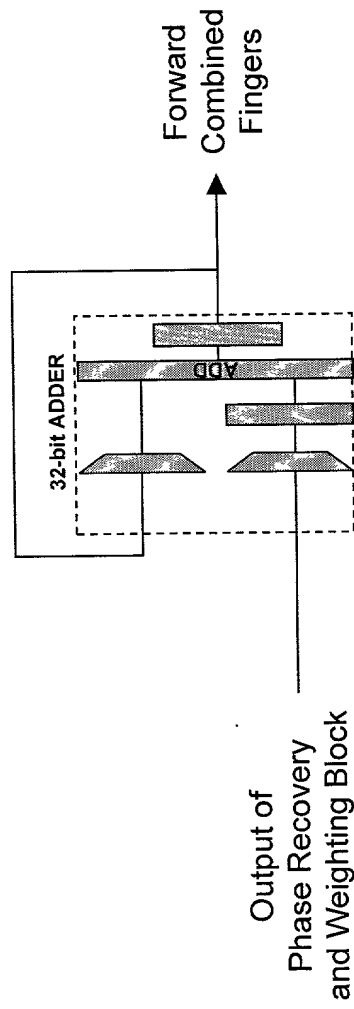
Finger Combiner Requirements and Assumptions

- Requirements
 - ◆ Must be able to combine up to six fingers
- Assumptions
 - ◆ Extra cycles can be wasted as long as the TPCG < 300us
 - ◆ Allocate timing such that the circuit is always adding 8 fingers
 - ◆ The ARC processor may assign up to 8 fingers to each user
 - ◆ The ARC processor assigns zeroes to the unused fingers in an eight finger block

Finger Combiner Functional Block Diagram



Finger Combiner Implementation



Finger Combiner Resource Requirements

- Implementation of:
 - 32 Users @ 125 MHz
 - 48 Users @ 187.5 MHz
 - 64 Users @ 250 MHz
- ◆ 1 DPU

General Timing and Control Finger Tracking within the Rake Receiver

- Upon initial acquisition, the Rake Receiver is tasked such that the Scrambling Code is set to an offset that places the (up to) six fingers in the Antenna Sample Buffer window such that zero path delay corresponds to the beginning of the buffer window
- As the mobile user moves toward or away from the base station (Node B), the fingers will move within the Antenna Sample Buffer window as tasked by the Path Searcher

Spreading Factor Mapping Assignments

- The following table lists the values corresponding to the various Spreading Factors (SF):
- Note that users requiring SF=4 must assign each finger to two successive finger resources per finger according to the following rules
 - ◆ At finger resource n assign SF=4
 - ◆ At finger resource n+1 assign SF=0 (used to denote second part of SF=4 finger)

SPREADING FACTOR	SF MEMORY CONTENTS
0	0
4	1
8	2
16	3
32	4
64	5
128	6
256	7

Chameleon Systems Rake Receiver CS2112 Device Utilization for 32 Users

KERNEL	DATA PATH UNITS (DPUs)	LOCAL STATE MEMORIES (LSMs)	MULTIPLIERS
Gold Code Generator	14	8	2
Channel Code Generator / Multiplier	4	2	0
Antenna Sample Buffer	22	26	0
Chip Descrambler	24	0	0
Chip Adder Tree	7	0	0
Chip Integrator	3	2	0
Channel Estimator Data Input Buffer	1	2	0
Channel Estimator UCC Input Buffer	1	2	0
Channel Estimator Output Buffer	1	2	0
Phase Recovery Input Buffer	3	3	0
Phase Recovery and Weighting	1	0	1
Finger Combiner	1	0	0
Master Chip Counter	2	0	0
TOTALS:	84	47	3
TOTAL AVAILABLE	84	48	24
PERCENT UTILIZED	100%	98%	13%

```

/*
$Id: galois.c,v 1.4 2000/03/21 14:49:27 rollins Exp $

galois.c

(c) 2000 Chameleon Systems Inc.
    Algorithms in Reconfigurable Silicon

Mark Rollins
14-Mar-2000
*/

#include "galois.h"

/*
-----
Polynomial Multiplication Modulo f(x) for GF(2^25)

This version is hard coded for GS(2^25).

Since we need to multiply polynomials of order 24, the result
will not fit in a 32-bit register. We need to manage two
such registers (upper and lower).

Polynomial multiplication is just simply shifts and adds
with the added complexity of managing the two registers.

Polynomial reduction modulo f(x) is performed by adding in
shifted correction terms f(x) which line up with the undesired
higher order 1's in the product (lying in the 25-th bit and above).
-----
*/

int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x )
{
    int lower, upper;
    int shft_a, shft_b, upper_b;
    int fsl, fsu;
    int i;

    /* Initialize */
    upper = 0;
    upper_b = 0;
    shft_a = a_x;
    shft_b = b_x;
    lower = ((shft_a & 1) == 1) ? b_x : 0; /* Shift 0 */

    /* Shifts 1 to 7 remain in lower register */
    for (i=1; i <= 7; i++) {
        shft_a >>= 1;
        shft_b <<= 1;
        if ((shft_a & 1) == 1)
            lower = lower ^ shft_b;
    }

    /* Shifts 8 to 24 spread across lower & upper registers */
    for (i=8; i <= 24; i++) {

```

```

shft_a >>= 1;
upper_b <=<= 1;
upper_b += ( (shft_b&0x80000000) >> 31 );
shft_b <=<= 1;
if ((shft_a & 1) == 1) {
    lower = lower ^ shft_b;
    upper = upper ^ upper_b;
}
}

/*
Perform modulo f(x) reduction on high order bits:
- check if bits 48, 47, ..., 25 are unity
- if yes, add shifted versions of f(x)
*/

/* Bits 48 to 32 spread across lower & upper registers */
fsu = f_x >> 9;
for (i=16; i>=0; i--) {
    fsl = f_x << (7+i);
    if ( ((upper>>i)&1) == 1 ) {
        upper = upper ^ fsu;
        lower = lower ^ fsl;
    }
    fsu >>= 1;
}

/* Bits 31 to 25 remain in lower register */
for (i=31; i >= 25; i--) {
    fsl = f_x << (i-25);
    if ( ((lower>>i)&1) == 1 )
        lower = lower ^ fsl;
}
/*    printf("Upper: %x\n", upper); */
/*    printf("Lower: %x\n", lower); */

return lower;
}

```

/*

Polynomial Division With Max Degree 25

Determine $a(x) = (g(x) / f(x))_{\deg < 25}$

where

$f(x)$ is a primitive polynomial of degree '25'
 $g(x)$ is a polynomial of degree < '25'
 $a(x)$ is a polynomial of degree < '25'

The higher order terms of $a(x)$ are not calculated.

*/

```

int poly_divide_max_degree_25( int g_x, int f_x )
{
    int i, a_x, shft_g;

```

```

a_x = 0;
shft_g = g_x;
for (i=0; i < 25; i++) {
    if ( (shft_g & 1) == 1 ) {
        shft_g = shft_g ^ f_x;
        a_x += (1 << i);
    }
    shft_g >>= 1;
}
return a_x;
}

```

/*

Polynomial Multiplication With Max Degree 25
Determine

$g(x) = (f(x)b(x))_{\text{deg}<25}$

where

$f(x)$ is a primitive polynomial of degree '25'
 $b(x)$ is a polynomial of degree < '25'
 $g(x)$ is restricted to have degree < '25'

The higher order terms of $g(x)$ are not calculated.

The primitive polynomial $f(x)$ is specified by its primitive polynomial.

The lower order polynomial terms are stored in LSB's.

*/

```

int poly_mult_max_degree_25( int a_x, int f_x )
{
    int i, g_x, shft_f;

    g_x = 0;
    shft_f = f_x;
    for (i=0; i <= 25; i++) {
        if ( (shft_f & 1) == 1 )
            g_x = g_x ^ ( a_x << i );
        shft_f >>= 1;
    }
    g_x = g_x & 0x1FFFFFFF;    /* Keep bits 0 through 24 */
    return g_x;
}

```

/*

Polynomial Multiplication With Max Degree 25 for UMTS Top Polynomial
Determine

$g(x) = (f(x)b(x))_{\text{deg}<25}$

where

$f(x)$ is a $1 + x^3 + x^{25}$
 $b(x)$ is a polynomial of degree < '25'
 $g(x)$ is restricted to have degree < '25'

The higher order terms of $g(x)$ are not calculated.

The primitive polynomial $f(x)$ is specified in the UMTS Standard.

The lower order polynomial terms are stored in LSB's.

*/

```
int poly_mult_max_degree_UMTS_top( int a_x )
{
    int g_x;

    g_x = a_x;
    g_x = g_x ^ ( a_x << 3 );
    g_x = g_x ^ ( a_x << 25 );
    g_x = g_x & 0x1FFFFFF;    /* Keep bits 0 through 24 */
    return g_x;
}
```

/*

Polynomial Multiplication With Max Degree 25 for UMTS Bottom Polynomial
Determine

$$g(x) = (f(x)b(x))_{\deg < 25}$$

where

$f(x)$ is a $1 + x + x^2 + x^3 + x^{25}$

$b(x)$ is a polynomial of degree $< '25'$

$g(x)$ is restricted to have degree $< '25'$

The higher order terms of $g(x)$ are not calculated.

The primitive polynomial $f(x)$ is specified in the UMTS Standard.

The lower order polynomial terms are stored in LSB's.

*/

```
int poly_mult_max_degree_UMTS_bot( int a_x )
{
    int g_x;

    g_x = a_x;
    g_x = g_x ^ ( a_x << 1 );
    g_x = g_x ^ ( a_x << 2 );
    g_x = g_x ^ ( a_x << 3 );
    g_x = g_x ^ ( a_x << 25 );
    g_x = g_x & 0x1FFFFFF;    /* Keep bits 0 through 24 */
    return g_x;
}
```

/*

Bit Reverse a 25-bit Integer

----- */

```
int bit_reverse_25( int g_x )
{
```

```
    int i, r_x;
```

```
    r_x = 0;
```



```

    for(i=0; i < 25; i++) {
        r_x <= 1;
        r_x += (g_x>>i) & 1;
    }
    return r_x;
}

/*
-----
LFSR Generator for N=25 with Mask Polynomial
-----
*/

#ifdef ARC
int LFSR_gen_25_mask( int f_x, int *a_x, int m_x )
{
    int i, lsb, rxor, new_msb;

    /* Calculate LSB using mask */
    rxor = m_x & *a_x;
    lsb = 0;
    for (i=0; i <= 24; i++) {
        lsb = lsb ^ (rxor & 1);
        rxor >>= 1;
    }

    /* Calculate NEW MSB */
    rxor = f_x & *a_x;
    new_msb = 0;
    for (i=0; i <= 24; i++) {
        new_msb = new_msb ^ (rxor & 1);
        rxor >>= 1;
    }

    /* Update state */
    *a_x = (*a_x>>1) ^ ( new_msb << 24 );

    return lsb;
}
#endif

/*
-----
LFSR Generator for N=25
-----
*/

#ifdef ARC
int LFSR_gen_25( int f_x, int *a_x )
{
    int i, lsb, rxor, new_msb;

    /* Extract LSB */
    lsb = *a_x & 1;

    /* Calculate NEW MSB */
    rxor = f_x & *a_x;

```

```

    new_msb = 0;
    for (i=0; i <= 24; i++) {
        new_msb = new_msb ^ (rxor & 1);
        rxor >>= 1;
    }

    /* Update state */
    *a_x = (*a_x>>1) ^ ( new_msb << 24 );

    return lsb;
}
#endif

/*
-----
Print Bitstring
where 'n' is the number of bits, 1 < n <= 32
-----
*/

#ifndef ARC
#include <stdio.h>

void print_bitstring( char *mesg, int poly, int n )
{
    int i;
    for(i=n-1; i >= 0; i--)
        printf("%1d ", (poly>>i) & 1);
    printf(mesg);
    printf("\n");
}
#endif

/*
-----
Reduction of x^power Modulo f(x) to a polynomial
where
    f(x) = 1 + x^3 + x^25
-----
*/

#ifndef ARC
#include <math.h>

int reduce_25( int power )
{
    int index, result, stop;
    short int *list = (short int*) calloc( power+1, sizeof(short int) );

    for (index=power; index >= 0; index--)
        list[index] = 0;

    list[power] = 1;
    index = power;

    while (index >= 25) {

```

```

        if (list[index]) {
            list[index] = 0;
            list[index-22] = list[index-22]^1;
            list[index-25] = list[index-25]^1;
        }
        index -= 1;
    }
    result = 0;
    stop = ( power < 25 ) ? power+1 : 25;
    for (index=0; index < stop; index++)
        result += (list[index] << index);
    free(list);
    return result;
}
#endif

/*
-----
Given a polynomial g(x), calculate the value of 'k' in

        g(x) = x^k modulo f(x)
-----
*/

#ifndef ARC
int revert_modulo_poly_reduction( int g_x, int f_x )
{
    int cnt = 0;
    int shft = g_x;
    while (shft != 1) {
        if ( (shft & 1) == 0 ) {
            do {
                shft >>= 1;
                cnt++;
            } while ( (shft & 1) == 0 );
        }
        else
            shft = shft ^ f_x;
    }
    return cnt;
}
#endif

/*
-----
Print a polynomial as a sum of powers of 'x'
-----
*/

#ifndef ARC
void print_poly_25( int g_x )
{
    int i, bit;
    for(i=0; i < 25; i++) {
        bit = (g_x>>i)&1;
        if (bit) {
            if (i==0)

```



```
/*  
$Id: galois.h,v 1.4 2000/03/21 14:49:36 rollins Exp $
```

```
galois.h
```

```
(c) 2000 Chameleon Systems Inc.  
Algorithms in Reconfigurable Silicon
```

```
Mark Rollins  
14-Mar-2000
```

```
*/
```

```
#ifndef _galios_h_
```

```
/* -----  
ARC Routines:  
----- */
```

```
int bit_reverse_25(int g_x);
```

```
int poly_mult_max_degree_UMTS_top( int a_x );
```

```
int poly_mult_max_degree_UMTS_bot( int a_x );
```

```
int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x );
```

```
int poly_divide_max_degree_25( int g_x, int f_x );
```

```
int poly_mult_max_degree_25( int a_x, int f_x );
```

```
/* -----  
Solaris/Debugging Routines:  
----- */
```

```
#ifndef ARC
```

```
int LFSR_gen_25_mask( int f_x, int *a_x, int m_x );
```

```
int LFSR_gen_25( int f_x, int *a_x );
```

```
void print_bitstring( char *mesg, int poly, int n );
```

```
int revert_modulo_poly_reduction( int g_x, int f_x );
```

```
void print_poly_25( int g_x );
```

```
int reduce_25( int power );
```

```
#endif
```

```
#define _galois_h_ 1
```

```
#endif
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187

```
/*
$Id: galois_arc.c,v 1.3 2000/03/21 00:19:27 rollins Exp $
```

```
galois_arc.c
```

```
(c) 2000 Chameleon Systems Inc.
    Algorithms in Reconfigurable Silicon
```

```
Mark Rollins
```

```
14-Mar-2000
```

```
*/
```

```
#include "galois.h"
```

```
#define N_bits 1000
```

```
#define mask 0x0040090
```

```
#define poly1 0x2000009
```

```
#define user 0x1000000
```

```
int main( int argc, char **argv )
```

```
{
```

```
    int alx, alx_rev, a2x, a2x_rev;
```

```
    int glx, g2x;
```

```
    alx_rev = user;
```

```
    /* Determine new seed required to produce a delayed
       version of the LFSR sequence
```

```
    */
```

```
    alx = bit_reverse_25( alx_rev );
```

```
    glx = poly_mult_max_degree_UMTS_top( alx );
```

```
    g2x = poly_mult_modulo_fx_2p25( glx, mask, poly1 );
```

```
    a2x = poly_divide_max_degree_25( g2x, poly1 );
```

```
    a2x_rev = bit_reverse_25( a2x );
```

```
}
```

Rollins, Mark
14-Mar-2000
galois_arc.c

```

/*
$Id: galois.h,v 1.4 2000/03/21 14:49:36 rollins Exp $

galois.h

(c) 2000 Chameleon Systems Inc.
    Algorithms in Reconfigurable Silicon

Mark Rollins
14-Mar-2000
*/

#ifndef _galios_h_

/* -----
   ARC Routines: ----- */

int bit_reverse_25(int g_x);

int poly_mult_max_degree_UMTS_top( int a_x );

int poly_mult_max_degree_UMTS_bot( int a_x );

int poly_mult_modulo_fx_2p25( int a_x, int b_x, int f_x );

int poly_divide_max_degree_25( int g_x, int f_x );

int poly_mult_max_degree_25( int a_x, int f_x );

/* -----
   Solaris/Debugging Routines: ----- */

#ifndef ARC

int LFSR_gen_25_mask( int f_x, int *a_x, int m_x );

int LFSR_gen_25( int f_x, int *a_x );

void print_bitstring( char *mesg, int poly, int n );

int revert_modulo_poly_reduction( int g_x, int f_x );

void print_poly_25( int g_x );

int reduce_25( int power );

#endif

#define _galois_h_ 1
#endif

```


**COMBINED DECLARATION AND POWER OF ATTORNEY
FOR DESIGN PATENT APPLICATION**

Attorney's Docket No.

032001-074

As a below-named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name;

I BELIEVE I AM THE ORIGINAL, FIRST AND SOLE INVENTOR (if only one name is listed below) OR AN ORIGINAL, FIRST AND JOINT INVENTOR (if more than one name is listed below) OF THE SUBJECT MATTER WHICH IS CLAIMED AND FOR WHICH A PATENT IS SOUGHT ON THE INVENTION ENTITLED:

Gold Code Generator Design

the specification of which

(check one)

☒ is attached hereto;

☐ was filed on _____ as

Application No. _____

and was amended on _____;
(if applicable)

I HAVE REVIEWED AND UNDERSTAND THE CONTENTS OF THE ABOVE-IDENTIFIED SPECIFICATION, INCLUDING THE CLAIMS, AS AMENDED BY ANY AMENDMENT REFERRED TO ABOVE;

I ACKNOWLEDGE THE DUTY TO DISCLOSE TO THE OFFICE ALL INFORMATION KNOWN TO ME TO BE MATERIAL TO PATENTABILITY AS DEFINED IN TITLE 37, CODE OF FEDERAL REGULATIONS, Sec. 1.56 (as amended effective March 16, 1992);

I do not know and do not believe the said invention was ever known or used in the United States of America before my or our invention thereof, or patented or described in any printed publication in any country before my or our invention thereof or more than one year prior to said application; that said invention was not in public use or on sale in the United States of America more than one year prior to said application; that said invention has not been patented or made the subject of an inventor's certificate issued before the date of said application in any country foreign to the United States of America on any application filed by me or my legal representatives or assigns more than six months prior to said application;

I hereby claim foreign priority benefits under Title 35, United States Code Sec. 119 and Sec. 172 of any foreign application(s) for patent or inventor's certificate as indicated below and have also identified below any foreign application for patent or inventor's certificate on this invention having a filing date before that of the application(s) on which priority is claimed:

COMBINED DECLARATION AND POWER OF ATTORNEY

Attorney's Docket No.

032001-074

COUNTRY/INTERNATIONAL	APPLICATION NUMBER	DATE OF FILING (day, month, year)	PRIORITY CLAIMED
			YES_ NO_
			YES_ NO_

I hereby appoint the following attorneys and agent(s) to prosecute said application and to transact all business in the Patent and Trademark Office connected therewith and to file, prosecute and to transact all business in connection with international applications directed to said invention:

William L. Mathis	17,337	R. Danny Huntington	27,903	Gerald F. Swiss	30,113
Robert S. Swecker	19,885	Eric H. Weisblatt	30,505	Charles F. Wieland III	33,096
Platon N. Mandros	22,124	James W. Peterson	26,057	Bruce T. Wieder	33,815
Benton S. Duffett, Jr.	22,030	Teresa Stanek Rea	30,427	Todd R. Walters	34,040
Norman H. Stepno	22,716	Robert E. Krebs	25,885	Ronni S. Jillions	31,979
Ronald L. Grudziecki	24,970	William C. Rowland	30,888	Harold R. Brown III	36,341
Frederick G. Michaud, Jr.	26,003	T. Gene Dillahunty	25,423	Allen R. Baum	36,086
Alan E. Kopecki	25,813	Patrick C. Keane	32,858	Steven M. duBois	35,023
Regis E. Slutter	26,999	B. Jefferson Boggs, Jr.	32,344	Brian P. O'Shaughnessy	32,747
Samuel C. Miller, III	27,360	William H. Benz	25,952	Kenneth B. Leffler	36,075
Robert G. Mukai	28,531	Peter K. Skiff	31,917	Fred W. Hathaway	32,236
George A. Hovanec, Jr.	28,223	Richard J. McGrath	29,195		
James A. LaBarre	28,632	Matthew L. Schneider	32,814		
E. Joseph Gess	28,510	Michael G. Savage	32,596		



21839

and: Joseph P. O'Malley, Reg. No. 36,226

Address all correspondence to:



21839

Robert E. Krebs

BURNS, DOANE, SWECKER & MATHIS, L.L.P.

P.O. Box 1404

Alexandria, Virginia 22313-1404

Address all telephone calls to: Joseph P. O'Malley

at (650)622-2300.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

FULL NAME OF SOLE OR FIRST INVENTOR	SIGNATURE	DATE
DANIEL J. PUGH	<i>Daniel J. Pugh</i>	10/26/00
RESIDENCE	CITIZENSHIP	
1595 Rebel Way, San Jose, CA 95118	United States	
POST OFFICE ADDRESS		
1595 Rebel Way, San Jose, CA 95118		
FULL NAME OF SECOND JOINT INVENTOR, IF ANY	SIGNATURE	DATE
MARK ROLLINS		
RESIDENCE	CITIZENSHIP	
23 Stonepath Crescent, Stittsville, Ontario, Canada	Canada	
POST OFFICE ADDRESS		
23 Stonepath Crescent, Stittsville, Ontario, Canada		